USENIX

Proceedings of the 12th USENIX Security Symposium

Washington, D.C., August 4–8, 2003

conference

proceedings

12th USENIX Security Symposium

Washington, D.C., USA

August 4–8, 2003

Sponsored by

The USENIX Association

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
URL: http://www.usenix.org.

Past USENIX Security Proceedings

Security XI	August 2002	Washington, D.C., USA	$30/38
Security X	August 2001	Washington, D.C., USA	$30/38
Security IX	August 2000	Denver, Colorado, USA	$27/35
Security VIII	August 1999	Washington, D.C., USA	$27/35
Security VII	January 1998	San Antonio, Texas, USA	$27/35
Security VI	July 1996	San Jose, California, USA	$27/35
Security V	June 1995	Salt Lake City, Utah, USA	$27/35
Security IV	October 1993	Santa Clara, California, USA	$15/20
Security III	September 1992	Baltimore, Maryland, USA	$30/39
Security II	August 1990	Portland, Oregon, USA	$13/16
Security	August 1988	Portland, Oregon, USA	$7/7

ISBN 1-931971-13-7

Printed in the United States of America on 50% recycled paper, 10–15% post-consumer waste.

USENIX Association

Proceedings of the 12th USENIX Security Symposium

August 4–8, 2003
Washington, D.C., USA

Symposium Organizers

Program Chair

Vern Paxson, *International Computer Science Institute*

Invited Talks Coordinator

Matt Blaze, *AT&T Labs—Research*

Program Committee

Steve Bellovin, *AT&T Labs—Research*
Dan Boneh, *Stanford University*
Crispin Cowan, *Immunix, Inc.*
Drew Dean, *SRI International*
Kevin Fu, *MIT*
Peter Gutmann, *University of Auckland, New Zealand*
Richard Kemmerer, *University of California, Santa Barbara*
Patrick McDaniel, *AT&T Labs—Research*
John McHugh, *CERT® Coordination Center*
Radia Perlman, *Sun Microsystems*
Niels Provos, *CITI, University of Michigan*
Dawn Song, *Carnegie Mellon University*
David Wagner, *University of California, Berkeley*
Dan S. Wallach, *Rice University*
Elizabeth Zwicky, *Counterpane Internet Security*

The USENIX Association Staff

External Reviewers

William Arbaugh
Seth Arnold
Derek Atkins
Steve Beattie
Tom Berson
Nikita Borisov
Eric Cronin
Andy Ellis
Matthew Franklin
Simson Garfinkel
Li Gong
Rachel Greenstadt
Pankaj Gupta
Matthias Jacob
Stas Jarecki
Christopher Kruegel
Peter Neumann
Alan Olsen
Seth Proctor
Zulfikar Ramzan
Volker Roth
Algis Rudys
Naveen Sastry
Hovav Shacham
Alex Snoeren
Robin Sommer
Ping Tao
Jason Waddle
Perry Wagle
Nicholas Weaver
Tara Whalen

12th USENIX Security Symposium

August 4–8, 2003
Washington, D.C., USA

Index of Authors

Message from the Symposium Chair

The 21 papers in these proceedings were presented at the 12th USENIX Security Symposium, held August 4–8, 2003, in Washington, D.C. The papers were selected from 128 submissions covering a broad range of topics in computer and network security from researchers worldwide. A program committee of 16 experts selected the papers on the basis of excellence of scientific contribution.

The selection process began just after the submission deadline. Each paper was assigned for review to program committee members (usually three) and evaluated based on scientific novelty, contribution to the field, and technical quality. The authors did not know the identities of the reviewers. For some papers advice was also sought from colleagues outside the program committee. We are grateful to William Arbaugh, Seth Arnold, Derek Atkins, Steve Beattie, Tom Berson, Nikita Borisov, Eric Cronin, Andy Ellis, Matthew Franklin, Simson Garfinkel, Li Gong, Rachel Greenstadt, Pankaj Gupta, Matthias Jacob, Stas Jarecki, Christopher Kruegel, Peter Neumann, Alan Olsen, Seth Proctor, Zulfikar Ramzan, Volker Roth, Algis Rudys, Naveen Sastry, Hovav Shacham, Alex Snoeren, Robin Sommer, Ping Tao, Jason Waddle, Perry Wagle, Nicholas Weaver, and Tara Whalen in this regard. Final selection of papers for the symposium took place at the program committee meeting on March 14, 2003, at the International Computer Science Institute in Berkeley.

In addition to the papers printed here, the symposium included tutorials, a series of invited talks, a keynote speech by "Black Unicorn," two panel discussions, and a work-in-progress session.

I am most grateful to the program committee members for their efforts in reviewing the large number of submissions, generating on average more than 100 lines of review per submission. I would also like to thank the many authors who submitted papers. An enormous amount of hard work goes into generating a quality program, not least by the many authors whose papers did not quite make it. I sincerely hope that all authors who submitted papers have found the reviewers' comments helpful.

Many people helped make this conference a success. I am especially grateful to the USENIX staff for all the effort they put into organizing and running the conference. In addition, their help throughout the reviewing process was indispensable. I would also like to thank Avi Rubin, who served as the liaison for the USENIX Board of Directors, helping with numerous issues that came up during the selection process.

Finally, I would like to thank Matt Blaze for organizing a particularly stimulating invited talks track, and Kevin Fu for putting together the likewise stimulating Work-in-Progress reports.

Vern Paxson, International Computer Science Institute
Security '03 Program Chair

Remote Timing Attacks are Practical

David Brumley
Stanford University
dbrumley@cs.stanford.edu

Dan Boneh
Stanford University
dabo@cs.stanford.edu

Abstract

Timing attacks are usually used to attack weak computing devices such as smartcards. We show that timing attacks apply to general software systems. Specifically, we devise a timing attack against OpenSSL. Our experiments show that we can extract private keys from an OpenSSL-based web server running on a machine in the local network. Our results demonstrate that timing attacks against network servers are practical and therefore security systems should defend against them.

1 Introduction

Timing attacks enable an attacker to extract secrets maintained in a security system by observing the time it takes the system to respond to various queries. For example, Kocher [10] designed a timing attack to expose secret keys used for RSA decryption. Until now, these attacks were only applied in the context of hardware security tokens such as smartcards [4, 10, 18]. It is generally believed that timing attacks cannot be used to attack general purpose servers, such as web servers, since decryption times are masked by many concurrent processes running on the system. It is also believed that common implementations of RSA (using Chinese Remainder and Montgomery reductions) are not vulnerable to timing attacks.

We challenge both assumptions by developing a remote timing attack against OpenSSL [15], an SSL library commonly used in web servers and other SSL applications. Our attack client measures the time an OpenSSL server takes to respond to decryption queries. The client is able to extract the private key stored on the server. The attack applies in several environments.

Network. We successfully mounted our timing attack between two machines on our campus network. The attacking machine and the server were in different buildings with three routers and multiple switches between them. With this setup we were able to extract the SSL private key from common SSL applications such as a web server (Apache+mod_SSL) and a SSL-tunnel.

Interprocess. We successfully mounted the attack between two processes running on the same machine. A hosting center that hosts two domains on the same machine might give management access to the admins of each domain. Since both domain are hosted on the same machine, one admin could use the attack to extract the secret key belonging to the other domain.

Virtual Machines. A Virtual Machine Monitor (VMM) is often used to enforce isolation between two Virtual Machines (VM) running on the same processor. One could protect an RSA private key by storing it in one VM and enabling other VM's to make decryption queries. For example, a web server could run in one VM while the private key is stored in a separate VM. This is a natural way of protecting secret keys since a break-in into the web server VM does not expose the private key. Our results show that when using OpenSSL the network server VM can extract the RSA private key from the secure VM, thus invalidating the isolation provided by the VMM. This is especially relevant to VMM projects such as Microsoft's NGSCB architecture (formerly Palladium). We also note that NGSCB enables an application to ask the VMM (aka Nexus) to decrypt (aka unseal) application data. The application could expose the VMM's secret key by measuring the time the VMM takes to respond to such requests.

Many crypto libraries completely ignore the timing attack and have no defenses implemented to prevent it. For example, libgcrypt [14] (used in GNUTLS and GPG) and Cryptlib [5] do not defend against timing attacks. OpenSSL 0.9.7 implements a defense against the timing attack as an option. However, common applications such as mod_SSL, the Apache SSL module, do not en-

able this option and are therefore vulnerable to the attack. These examples show that timing attacks are a largely ignored vulnerability in many crypto implementations. We hope the results of this paper will help convince developers to implement proper defenses (see Section 6). Interestingly, Mozilla's NSS crypto library properly defends against the timing attack. We note that most crypto acceleration cards also implement defenses against the timing attack. Consequently, network servers using these accelerator cards are not vulnerable.

We chose to tailor our timing attack to OpenSSL since it is the most widely used open source SSL library. The OpenSSL implementation of RSA is highly optimized using Chinese Remainder, Sliding Windows, Montgomery multiplication, and Karatsuba's algorithm. These optimizations cause both known timing attacks on RSA [10, 18] to fail in practice. Consequently, we had to devise a new timing attack based on [18, 19, 20, 21, 22] that is able to extract the private key from an OpenSSL-based server. As we will see, the performance of our attack varies with the exact environment in which it is applied. Even the exact compiler optimizations used to compile OpenSSL can make a big difference.

In Sections 2 and 3 we describe OpenSSL's implementation of RSA and the timing attack on OpenSSL. In Section 4 we discuss how these attacks apply to SSL. In Section 5 we describe the actual experiments we carried out. We show that using about a million queries we can remotely extract a 1024-bit RSA private key from an OpenSSL 0.9.7 server. The attack takes about two hours.

Timing attacks are related to a class of attacks called side-channel attacks. These include power analysis [9] and attacks based on electromagnetic radiation [16]. Unlike the timing attack, these extended side channel attacks require special equipment and physical access to the machine. In this paper we only focus on the timing attack. We also note that our attack targets the implementation of RSA decryption in OpenSSL. Our timing attack does not depend upon the RSA padding used in SSL and TLS.

2 OpenSSL's Implementation of RSA

We begin by reviewing how OpenSSL implements RSA decryption. We only review the details needed for our attack. OpenSSL closely follows algorithms described in the Handbook of Applied Cryptography [11], where more information is available.

2.1 OpenSSL Decryption

At the heart of RSA decryption is a modular exponentiation $m = c^d \bmod N$ where $N = pq$ is the RSA modulus, d is the private decryption exponent, and c is the ciphertext being decrypted. OpenSSL uses the Chinese Remainder Theorem (CRT) to perform this exponentiation. With Chinese remaindering, the function $m = c^d \bmod N$ is computed in two steps. First, evaluate $m_1 = c^{d_1} \bmod p$ and $m_2 = c^{d_2} \bmod q$ (here d_1 and d_2 are precomputed from d). Then, combine m_1 and m_2 using CRT to yield m.

RSA decryption with CRT gives up to a factor of four speedup, making it essential for competitive RSA implementations. RSA with CRT is not vulnerable to Kocher's original timing attack [10]. Nevertheless, since RSA with CRT uses the factors of N, a timing attack can expose these factors. Once the factorization of N is revealed it is easy to obtain the decryption key by computing $d = e^{-1} \bmod (p-1)(q-1)$.

2.2 Exponentiation

During an RSA decryption with CRT, OpenSSL computes $c^{d_1} \bmod p$ and $c^{d_2} \bmod q$. Both computations are done using the same code. For simplicity we describe how OpenSSL computes $g^d \bmod q$ for some g, d, and q.

The simplest algorithm for computing $g^d \bmod q$ is *square and multiply*. The algorithm squares g approximately $\log_2 d$ times, and performs approximately $\frac{\log_2 d}{2}$ additional multiplications by g. After each step, the product is reduced modulo q.

OpenSSL uses an optimization of square and multiply called *sliding windows* exponentiation. When using sliding windows a block of bits (window) of d are processed at each iteration, where as simple square-and-multiply processes only one bit of d per iteration. Sliding windows requires pre-computing a multiplication table, which takes time proportional to $2^{w-1}+1$ for a window of size w. Hence, there is an optimal window size that balances the time spent during precomputation vs. actual exponentiation. For a 1024-bit modulus OpenSSL uses a window size of five so that about five bits of the exponent d are processed in every iteration.

For our attack, the key fact about sliding windows is that during the algorithm there are many multiplications by g, where g is the input ciphertext. By querying on many

inputs g the attacker can expose information about bits of the factor q. We note that a timing attack on sliding windows is much harder than a timing attack on square-and-multiply since there are far fewer multiplications by g in sliding windows. As we will see, we had to adapt our techniques to handle sliding windows exponentiation used in OpenSSL.

2.3 Montgomery Reduction

The sliding windows exponentiation algorithm performs a modular multiplication at every step. Given two integers x, y, computing $xy \bmod q$ is done by first multiplying the integers $x * y$ and then reducing the result modulo q. Later we will see each reduction also requires a few additional multiplications. We first briefly describe OpenSSL's modular reduction method and then describe its integer multiplication algorithm.

Naively, a reduction modulo q is done via multi-precision division and returning the remainder. This is quite expensive. In 1985 Peter Montgomery discovered a method for implementing a reduction modulo q using a series of operations efficient in hardware and software [13].

Montgomery reduction transforms a reduction modulo q into a reduction modulo some power of 2 denoted by R. A reduction modulo a power of 2 is faster than a reduction modulo q as many arithmetic operations can be implemented directly in hardware. However, in order to use Montgomery reduction all variables must first be put into Montgomery form. The Montgomery form of number x is simply $xR \bmod q$. To multiply two numbers a and b in Montgomery form we do the following. First, compute their product as integers: $aR*bR = cR^2$. Then, use the fast Montgomery reduction algorithm to compute $cR^2 * R^{-1} = cR \bmod q$. Note that the result $cR \bmod q$ is in Montgomery form, and thus can be directly used in subsequent Montgomery operations. At the end of the exponentiation algorithm the output is put back into standard (non-Montgomery) form by multiplying it by $R^{-1} \bmod q$. For our attack, it is equivalent to use R and $R^{-1} \bmod N$, which are public.

Hence, for the small penalty of converting the input g to Montgomery form, a large gain is achieved during modular reduction. With typical RSA parameters the gain from Montgomery reduction outweighs the cost of initially putting numbers in Montgomery form and converting back at the end of the algorithm.

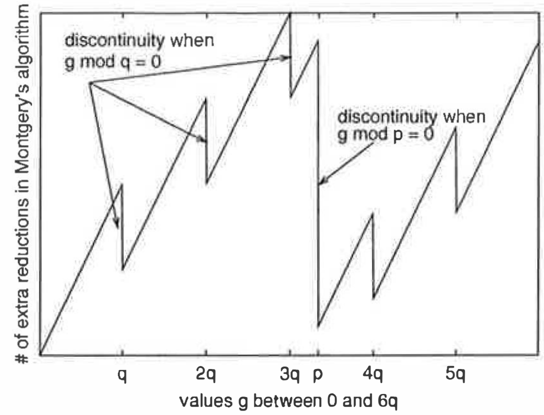


Figure 1: Number of extra reductions in a Montgomery reduction as a function (equation 1) of the input g.

The key relevant fact about a Montgomery reduction is at the end of the reduction one checks if the output cR is greater than q. If so, one subtracts q from the output, to ensure that the output cR is in the range $[0, q)$. This extra step is called an *extra reduction* and causes a timing difference for different inputs. Schindler noticed that the probability of an extra reduction during an exponentiation $g^d \bmod q$ is proportional to how close g is to q [18]. Schindler showed that the probability for an extra reduction is:

$$\Pr[\texttt{Extra Reduction}] = \frac{g \bmod q}{2R} \tag{1}$$

Consequently, as g approaches either factor p or q from below, the number of extra reductions during the exponentiation algorithm greatly increases. At exact multiples of p or q, the number of extra reductions drops dramatically. Figure 1 shows this relationship, with the discontinuities appearing at multiples of p and q. By detecting timing differences that result from extra reductions we can tell how close g is to a multiple of one of the factors.

2.4 Multiplication Routines

RSA operations, including those using Montgomery's method, must make use of a multi-precision integer multiplication routine. OpenSSL implements two multiplication routines: Karatsuba (sometimes called recursive) and "normal". Multi-precision libraries represent large integers as a sequence of words. OpenSSL uses Karatsuba multiplication when multiplying two numbers with an equal number of words. Karatsuba multiplication takes time $O(n^{\log_2 3})$ which is $O(n^{1.58})$. OpenSSL uses

normal multiplication, which runs in time $O(nm)$, when multiplying two numbers with an unequal number of words of size n and m. Hence, for numbers that are approximately the same size (i.e. n is close to m) normal multiplication takes quadratic time.

Thus, OpenSSL's integer multiplication routine leaks important timing information. Since Karatsuba is typically faster, multiplication of two unequal size words takes longer than multiplication of two equal size words. Time measurements will reveal how frequently the operands given to the multiplication routine have the same length. We use this fact in the timing attack on OpenSSL.

In both algorithms, multiplication is ultimately done on individual words. The underlying word multiplication algorithm dominates the total time for a decryption. For example, in OpenSSL the underlying word multiplication routine typically takes $30\% - 40\%$ of the total runtime. The time to multiply individual words depends on the number of bits per word. As we will see in experiment 3 the exact architecture on which OpenSSL runs has an impact on timing measurements used for the attack. In our experiments the word size was 32 bits.

2.5 Comparison of Timing Differences

So far we identified two algorithmic data dependencies in OpenSSL that cause time variance in RSA decryption: (1) Schindler's observation on the number of extra reductions in a Montgomery reduction, and (2) the timing difference due to the choice of multiplication routine, i.e. Karatsuba vs. normal. Unfortunately, the effects of these optimizations counteract one another.

Consider a timing attack where we decrypt a ciphertext g. As g approaches a multiple of the factor q from below, equation (1) tells us that the number of extra reductions in a Montgomery reduction increases. When we are just over a multiple of q, the number of extra reductions decreases dramatically. In other words, decryption of $g < q$ should be slower than decryption of $g > q$.

The choice of Karatsuba vs. normal multiplication has the opposite effect. When g is just below a multiple of q, then OpenSSL almost always uses fast Karatsuba multiplication. When g is just over a multiple of q then $g \bmod q$ is small and consequently most multiplications will be of integers with different lengths. In this case, OpenSSL uses normal multiplication which is slower. In other words, decryption of $g < q$ should be faster than decryption of $g > q$ — the exact opposite of the effect of extra reductions in Montgomery's algorithm. Which effect dominates is determined by the exact environment. Our attack uses both effects, but each effect is dominant at a different phase of the attack.

3 A Timing Attack on OpenSSL

Our attack exposes the factorization of the RSA modulus. Let $N = pq$ with $q < p$. We build approximations to q that get progressively closer as the attack proceeds. We call these approximations guesses. We refine our guess by learning bits of q one at a time, from most significant to least. Thus, our attack can be viewed as a binary search for q. After recovering the half-most significant bits of q, we can use Coppersmith's algorithm [3] to retrieve the complete factorization.

Initially our guess g of q lies between 2^{512} (i.e. $2^{\log_2 N/2}$) and 2^{511} (i.e. $2^{\log_2 (N/2)-1}$). We then time the decryption of all possible combinations of the top few bits (typically 2-3). When plotted, the decryption times will show two peaks: one for q and one for p. We pick the values that bound the first peak, which in OpenSSL will always be q.

Suppose we already recovered the top $i-1$ bits of q. Let g be an integer that has the same top $i-1$ bits as q and the remaining bits of g are 0. Then $g < q$. At a high level, we recover the i'th bit of q as follows:

- Step 1 - Let g_{hi} be the same value as g, with the i'th bit set to 1. If bit i of q is 1, then $g < g_{hi} < q$. Otherwise, $g < q < g_{hi}$.
- Step 2 - Compute $u_g = gR^{-1} \bmod N$ and $u_{g_{hi}} = g_{hi}R^{-1} \bmod N$. This step is needed because RSA decryption with Montgomery reduction will calculate $u_g R = g$ and $u_{g_{hi}} R = g_{hi}$ to put u_g and $u_{g_{hi}}$ in Montgomery form before exponentiation during decryption.
- Step 3 We measure the time to decrypt both u_g and $u_{g_{hi}}$. Let t_1 = DecryptTime(u_g) and t_2 = DecryptTime($u_{g_{hi}}$).
- Step 4 - We calculate the difference $\Delta = |t_1 - t_2|$. If $g < q < g_{hi}$ then, by Section 2.5, the difference Δ will be "large", and bit i of q is 0. If $g < g_{hi} < q$, the difference Δ will be "small", and bit i of q is 1. We use previous Δ values to know what to consider "large" and "small". Thus we use the value $|t_1 - t_2|$ as an indicator for the i'th bit of q.

When the i'th bit is 0, the "large" difference can either be negative or positive. In this case, if $t_1 - t_2$ is positive then DecryptTime(g) > DecryptTime(g_{hi}), and the Montgomery reductions dominated the time difference. If $t_1 - t_2$ is negative, then DecryptTime(g) < DecryptTime(g_{hi}), and the multi-precision multiplication dominated the time difference.

Formatting of RSA plaintext, e.g. PKCS 1, does not affect this timing attack. We also do not need the value of the decryption, only how long the decryption takes.

3.1 Exponentiation Revisited

We would like $|t_{g_1} - t_{g_2}| \gg |t_{g_3} - t_{g_4}|$ when $g_1 < q < g_2$ and $g_3 < g_4 < q$. Time measurements that have this property we call a strong indicator for bits of q, and those that do not are a weak indicator for bits of q. Square and multiply exponentiation results in a strong indicator because there are approximately $\frac{\log_2 d}{2}$ multiplications by g during decryption. However, in sliding windows with window size w ($w = 5$ in OpenSSL) the expected number of multiplications by g is only:

$$E[\texttt{\# multiply by g}] \approx \frac{\log_2 d}{2^{w-1}(w+1)}$$

resulting in a weak indicator.

To overcome this we query at a *neighborhood* of values $g, g+1, g+2, ..., g+n$, and use the result as the decrypt time for g (and similarly for g_{hi}). The total decryption time for g or g_{hi} is then:

$$T_g = \sum_{i=0}^{n} \texttt{DecryptTime}(g+i)$$

We define T_g as the time to compute g with sliding windows when considering a neighborhood of values. As n grows, $|T_g - T_{g_{hi}}|$ typically becomes a stronger indicator for a bit of q (at the cost of additional decryption queries).

4 Real-world scenarios

As mentioned in the introduction there are a number of scenarios where the timing attack applies to networked servers. We discuss an attack on SSL applications, such as stunnel [23] and an Apache web server with mod_SSL [12], and an attack on trusted computing projects such as Microsoft's NGSCB (formerly Palladium).

During a standard full SSL handshake the SSL server performs an RSA decryption using its private key. The SSL server decryption takes place after receiving the CLIENT-KEY-EXCHANGE message from the SSL client. The CLIENT-KEY-EXCHANGE message is composed on the client by encrypting a PKCS 1 padded random bytes with the server's public key. The randomness encrypted by the client is used by the client and server to compute a shared master secret for end-to-end encryption.

Upon receiving a CLIENT-KEY-EXCHANGE message from the client, the server first decrypts the message with its private key and then checks the resulting plaintext for proper PKCS 1 formatting. If the decrypted message is properly formatted, the client and server can compute a shared master secret. If the decrypted message is not properly formatted, the server generates its own random bytes for computing a master secret and continues the SSL protocol. Note that an improperly formatted CLIENT-KEY-EXCHANGE message prevents the client and server from computing the same master secret, ultimately leading the server to send an ALERT message to the client indicating the SSL handshake has failed.

In our attack, the client substitutes a properly formatted CLIENT-KEY-EXCHANGE message with our guess g. The server decrypts g as a normal CLIENT-KEY-EXCHANGE message, and then checks the resulting plaintext for proper PKCS 1 padding. Since the decryption of g will not be properly formatted, the server and client will not compute the same master secret, and the client will ultimately receive an ALERT message from the server. The attacking client computes the time difference from sending g as the CLIENT-KEY-EXCHANGE message to receiving the response message from the server as the time to decrypt g. The client repeats this process for each value of of g and g_{hi} needed to calculate T_g and $T_{g_{hi}}$.

Our experiments are also relevant to trusted computing efforts such as NGSCB. One goal of NGSCB is to provide sealed storage. Sealed storage allows an application to encrypt data to disk using keys unavailable to the user. The timing attack shows that by asking NGSCB to decrypt data in sealed storage a user may learn the secret application key. Therefore, it is essential that the secure storage mechanism provided by projects such as NGSCB defend against this timing attack.

As mentioned in the introduction, RSA applications (and subsequently SSL applications using RSA for key exchange) using a hardware crypto accelerator are not vulnerable since most crypto accelerators implement defenses against the timing attack. Our attack applies to software based RSA implementations that do not defend against timing attacks as discussed in section 6.

5 Experiments

We performed a series of experiments to demonstrate the effectiveness of our attack on OpenSSL. In each case we show the factorization of the RSA modulus N is vulnerable. We show that a number of factors affect the efficiency of our timing attack.

Our experiments consisted of:

1. Test the effects of increasing the number of decryption requests, both for the same ciphertext and a neighborhood of ciphertexts.
2. Compare the effectiveness of the attack based upon different keys.
3. Compare the effectiveness of the attack based upon machine architecture and common compile-time optimizations.
4. Compare the effectiveness of the attack based upon source-based optimizations.
5. Compare inter-process vs. local network attacks.
6. Compare the effectiveness of the attack against two common SSL applications: an Apache web server with mod_SSL and stunnel.

The first four experiments were carried out inter-process via TCP, and directly characterize the vulnerability of OpenSSL's RSA decryption routine. The fifth experiment demonstrates our attack succeeds on the local network. The last experiment demonstrates our attack succeeds on the local network against common SSL-enabled applications.

5.1 Experiment Setup

Our attack was performed against OpenSSL 0.9.7, which does not blind RSA operations by default. All tests were run under RedHat Linux 7.3 on a 2.4 GHz Pentium 4 processor with 1 GB of RAM, using gcc 2.96 (RedHat). All keys were generated at random via OpenSSL's key generation routine.

For the first 5 experiments we implemented a simple TCP server that read an ASCII string, converted the string to OpenSSL's internal multi-precision representation, then performed the RSA decryption. The server returned 0 to signify the end of decryption. The TCP client measured the time from writing the ciphertext over the socket to receiving the reply.

Our timing attack requires a clock with fine resolution. We use the Pentium cycle counter on the attacking machine as such a clock, giving us a time resolution of 2.4 billion ticks per second. The cycle counter increments once per clock tick, regardless of the actual instruction issued. Thus, the decryption time is the cycle counter difference between sending the ciphertext to receiving the reply. The cycle counter is accessible via the "rdtsc" instruction, which returns the 64-bit cycle count since CPU initialization. The high 32 bits are returned into the EDX register, and the low 32 bits into the EAX register. As recommended in [7], we use the "cpuid" instruction to serialize the processor to prevent out-of-order execution from changing our timing measurements. Note that cpuid and rdtsc are only used by the attacking client, and that neither instruction is a privileged operation. Other architectures have a similar a counter, such as the UltraSparc %tick register.

OpenSSL generates RSA moduli $N = pq$ where $q < p$. In each case we target the smaller factor, q. Once q is known, the RSA modulus is factored and, consequently, the server's private key is exposed.

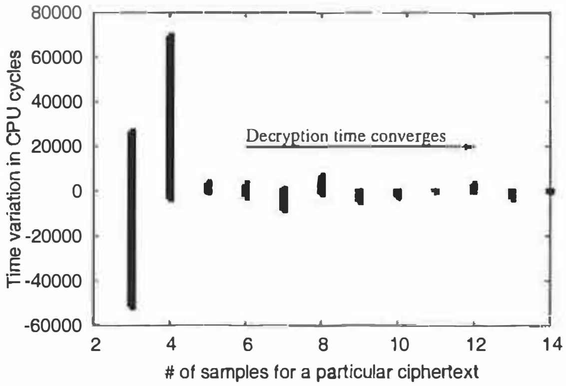
5.2 Experiment 1 - Number of Ciphertexts

This experiment explores the parameters that determine the number of queries needed to expose a single bit of an RSA factor. For any particular bit of q, the number of queries for guess g is determined by two parameters: neighborhood size and sample size.

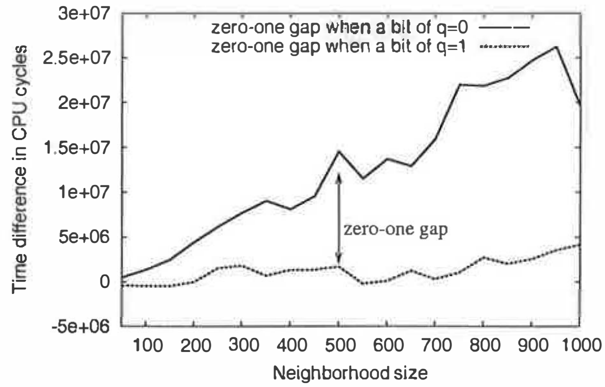
Neighborhood size. For every bit of q we measure the decryption time for a neighborhood of values $g, g+1, g+2, \ldots, g+n$. We denote this neighborhood size by n.

Sample size. For each value $g + i$ in a neighborhood we sample the decryption time multiple times and compute the mean decryption time. The number of times we query on each value $g + i$ is called the sample size and is denoted by s.

The total number of queries needed to compute T_g is then $s * n$.



(a) The time variance for decrypting a particular ciphertext decreases as we increase the number of samples taken.



(b) By increasing the neighborhood size we increase the zero-one gap between a bit of q that is 0 and a bit of q that is 1.

Figure 2: Parameters that affect the number of decryption queries of g needed to guess a bit of the RSA factor.

To overcome the effects of a multi-user environment, we repeatedly sample $g+k$ and use the median time value as the effective decryption time. Figure 2(a) shows the difference between median values as sample size increases. The number of samples required to reach a stable decryption time is surprising small, requiring only 5 samples to give a variation of under 20000 cycles (approximately 8 microseconds), well under that needed to perform a successful attack.

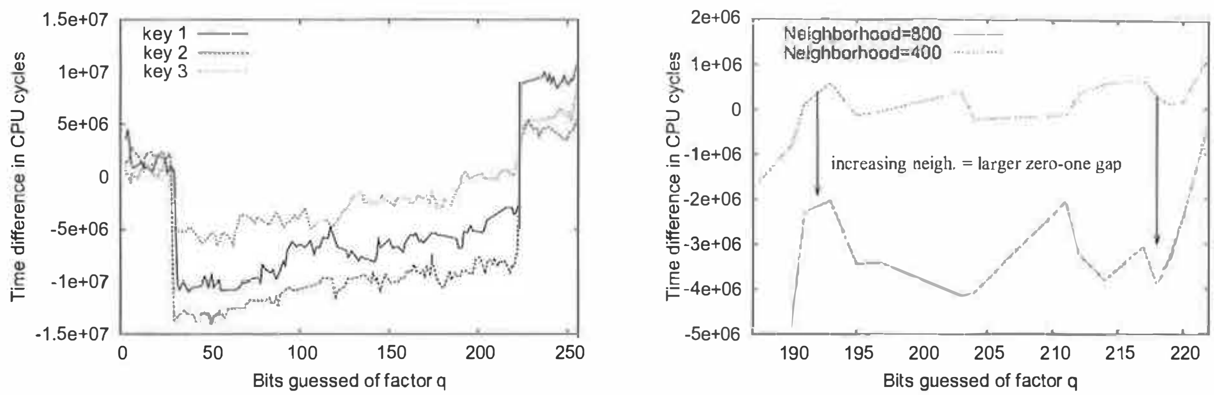
We call the gap between when a bit of q is 0 and 1 the *zero-one* gap. This gap is related to the difference $|T_g - T_{g_{hi}}|$, which we expect to be large when a bit of q is 0 and small otherwise. The larger the gap, the stronger the indicator that bit i is 0, and the smaller chance of error. Figure 2(b) shows that increasing the neighborhood size increases the size of the zero-one gap when a bit of q is 0, but is steady when a bit of q is 1.

The total number of queries to recover a factor is $2ns * \log_2 N/4$, where N is the RSA public modulus. Unless explicitly stated otherwise, we use a sample size of 7 and a neighborhood size of 400 on all subsequent experiments, resulting in 1433600 total queries. With these parameters a typical attack takes approximately 2 hours. In practice, an effective attack may need far fewer samples, as the neighborhood size can be adjusted dynamically to give a clear zero-one gap in the smallest number of queries.

5.3 Experiment 2 - Different Keys

We attacked several 1024-bit keys, each randomly generated, to determine the ease of breaking different moduli. In each case we were able to recover the factorization of N. Figure 3(a) shows our results for 3 different keys. For clarity, we include only bits of q that are 0, as bits of q that are 1 are close to the x-axis. In all our figures the time difference $T_g - T_{g_{hi}}$ is the zero-one gap. When the zero-one gap for bit i is far from the x-axis we can correctly deduce that bit i is 0.

With all keys the zero-one gap is positive for about the first 32 bits due to Montgomery reductions, since both g and g_{hi} use Karatsuba multiplication. After bit 32, the difference between Karatsuba and normal multiplication dominate until overcome by the sheer size difference between $\log_2(g \bmod q) - \log_2(g_{hi} \bmod q)$. The size difference alters the zero-one gaps because as bits of q are guessed, g_{hi} becomes smaller while g remains $\approx \log_2 q$. The size difference counteracts the effects of Karatsuba vs. normal multiplication. Normally the resulting zero-one gap shift happens around multiples of 32 (224 for key 1, 191 for key 2 and 3), our machine word size. Thus, an attacker should be aware that the zero-one gap may flip signs when guessing bits that are around multiples of the machine word size.



(a) The zero-one gap $T_g - T_{g_{hi}}$ indicates that we can distinguish between bits that are 0 and 1 of the RSA factor q for 3 different randomly-generated keys. For clarity, bits of q that are 1 are omitted, as the x-axis can be used for reference for this case.

(b) When the neighborhood is 400, the zero-one gap is small for some bits in key 3, making it difficult to distinguish between the 0 and 1 bits of q. By increasing the neighborhood size to 800, the zero-one gap is increased and we can launch a successful attack.

Figure 3: Breaking 3 RSA Keys by looking at the zero-one gap time difference

As discussed previously we can increase the size of the neighborhood to increase $|T_g - T_{g_{hi}}|$, giving a stronger indicator. Figure 3(b) shows the effects of increasing the neighborhood size from 400 to 800 to increase the zero-one gap, resulting in a strong enough indicator to mount a successful attack on bits 190-220 of q in key 3.

The results of this experiment show that the factorization of each key is exposed by our timing attack by the zero-one gap created by the difference when a bit of q is 0 or 1. The zero-one gap can be increased by increasing the neighborhood size if hard-to-guess bits are encountered.

5.4 Experiment 3 - Architecture and Compile-Time Effects

In this experiment we show how the computer architecture and common compile-time optimizations can affect the zero-one gap in our attack. Previously, we have shown how algorithmically the number of extra Montgomery reductions and whether normal or Karatsuba multiplication is used results in a timing attack. However, the exact architecture on which decryption is performed can change the zero-one gap.

To show the effect of architecture on the timing attack, we begin by showing the total number of instructions retired agrees with our algorithmic analysis of OpenSSL's decryption routines. An instruction is retired when it completes and the results are written to the destination [8]. However, programs with similar retirement counts may have different execution profiles due to different run-time factors such as branch predictions, pipeline throughput, and the L1 and L2 cache behavior.

We show that minor changes in the code can change the timing attack in two programs: "regular" and "extra-inst". Both programs time local calls to the OpenSSL decryption routine, i.e. unlike other programs presented "regular" and "extra-inst" are not network clients attacking a network server. The "extra-inst" is identical to "regular" except 6 additional nop instructions inserted before timing decryptions. The nop's only change subsequent code offsets, including those in the linked OpenSSL library.

Table 1 shows the timing attack with both programs for two bits of q. Montgomery reductions cause a positive instruction retired difference for bit 30, as expected. The difference between Karatsuba and normal multiplication cause a negative instruction retired difference for bit 32, again as expected. However, the difference $T_g - T_{g_{hi}}$ does not follow the instructions retired difference. On bit 30, there is about a 4 million extra cycles difference between the "regular" and "extra-inst" programs, even though the instruction retired count decreases. For bit 32, the change is even more pronounced: the zero-one gap changes sign between the "normal" and "extra-inst" programs while the instructions retired are similar!

	$g - g_{hi}$ retired	$T_g - T_{g_{hi}}$ cycles
"regular" bit 30	4579248 (0.009%)	6323188 (0.057%)
"extra-inst" bit 30	7641653 (0.016%)	2392299 (0.022%)
"regular" bit 32	-14275879 (-0.029%)	-5429545 (-0.049%)
"extra-inst" bit 32	-13187257 (-0.027%)	1310809 (0.012%)

Table 1: Bit 30 of q for both "regular" and "extra-inst" (which has a few additional nop's) have a positive instructions retired difference due to Montgomery reductions. Similarly, bit 32 has a negative instruction difference due to normal vs. Karatsuba multiplication. However, the addition of a few nop instructions in the "extra-instr" program changes the timing profile, most notably for bit 32. The percentages given are the difference divided by either the total of instructions retired or cycles as appropriate.

Extensive profiling using Intel's VTune [6] shows no single cause for the timing differences. However, two of the most prevalent factors were the L1 and L2 cache behavior and the number of instructions speculatively executed incorrectly. For example, while the "regular" program suffers approximately 0.139% L1 and L2 cache misses per load from memory on average, "extra-inst" has approximately 0.151% L1 and L2 cache misses per load. Additionally, the "regular" program speculatively executed about 9 million micro-operations incorrectly. Since the timing difference detected in our attack is only about 0.05% of total execution time, we expect the runtime factors to heavily affect the zero-one gap. However, under normal circumstances some zero-one gap should be present due to the input data dependencies during decryption.

The total number of decryption queries required for a successful attack also depends upon how OpenSSL is compiled. The compile-time optimizations change both the number of instructions, and how efficiently instructions are executed on the hardware. To test the effects of compile-time optimizations, we compiled OpenSSL three different ways:

- **Optimized** (-O3 -fomit-frame-pointer -mcpu=pentium): The default OpenSSL flags for Intel. -O3 is the optimization level, -fomit-frame-pointer omits the frame pointer, thus freeing up an extra register, and -mcpu=pentium enables more sophisticated resource scheduling.

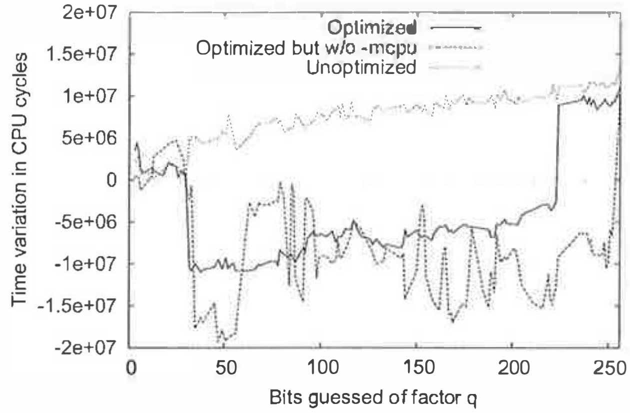


Figure 4: Different compile-time flags can shift the zero-one gap by changing the resulting code and how efficiently it can be executed.

- **No Pentium flag** (-O3 -fomit-frame-pointer): The same as the above, but without -mcpu sophisticated resource scheduling is not done, and an i386 architecture is assumed.
- **Unoptimized** (-g): Enable debugging support.

Each different compile-time optimization changed the zero-one gap. Figure 4 compares the results of each test. For readability, we only show the difference $T_g - T_{g_{hi}}$ when bit i of q is 0 ($g < q < g_{hi}$). The case where bit $i = 1$ shows little variance based upon the optimizations, and the x-axis can be used for reference.

Recall we expected Montgomery reductions to dominate when guessing the first 32 bits (with a positive zero-one gap), switching to Karatsuba vs. normal multiplication (with a negative zero-one gap) thereafter. Surprisingly, the unoptimized OpenSSL is unaffected by the Karatsuba vs. normal multiplication. Another surprising difference is the zero-one gap is more erratic when the -mcpu flag is omitted.

In these tests we again made about 1.4 million decryption queries. We note that without optimizations (-g), separate tests allowed us to recover the factorization with less than 359000 queries. This number could be reduced further by dynamically reducing the neighborhood size as bits of q are learned. Also, our tests of OpenSSL 0.9.6g were similar to the results of 0.9.7, suggesting previous versions of OpenSSL are also vulnerable.

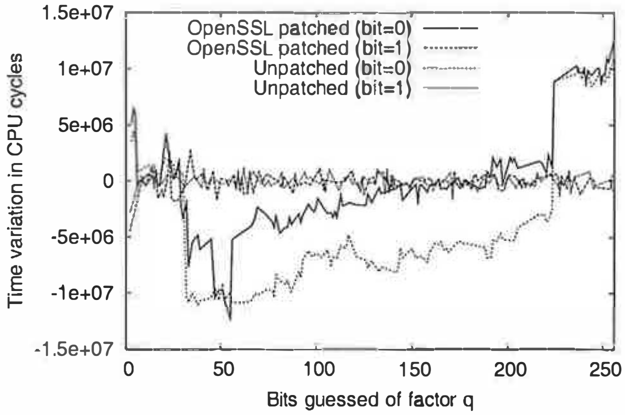


Figure 5: Minor source-based optimizations change the zero-one gap as well. As a consequence, code that doesn't appear initially vulnerable may become so as the source is patched.

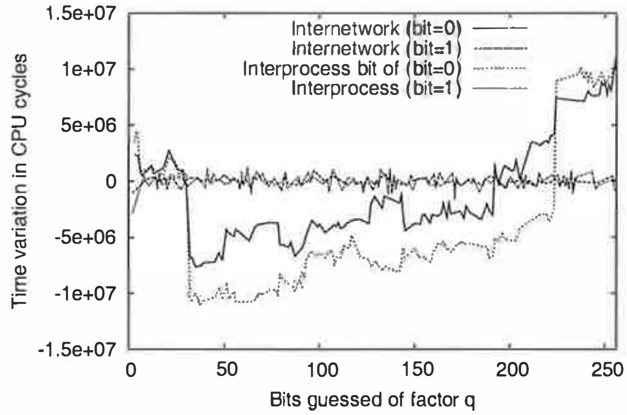


Figure 6: The timing attack succeeds over a local network. We contrast our results with the attack interprocess.

One conclusion we draw is that users of binary crypto libraries may find it hard to characterize their risk to our attack without complete understanding of the compile-time options and exact execution environment. Common flags such as enabling debugging support allow our attack to recover the factors of a 1024-bit modulus in about 1/3 million queries. We speculate that less complex architectures will be less affected by minor code changes, and have the zero-one gap as predicted by the OpenSSL algorithm analysis.

5.5 Experiment 4 - Source-based Optimizations

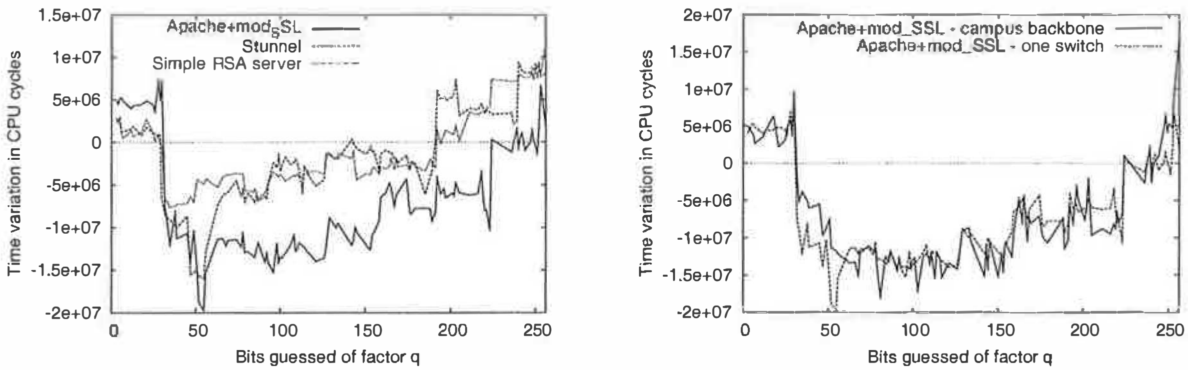
Source-based optimizations can also change the zero-one gap. RSA library developers may believe their code is not vulnerable to the timing attack based upon testing. However, subsequent patches may change the code profile resulting in a timing vulnerability. To show that minor source changes also affect our attack, we implemented a minor patch that improves the efficiency of the OpenSSL 0.9.7 CRT decryption check. Our patch has been accepted for future incorporation to OpenSSL (tracking ID 475).

After a CRT decryption, OpenSSL re-encrypts the result (mod N) and verifies the result is identical to the original ciphertext. This verification step prevents an incorrect CRT decryption from revealing the factors of the modulus [2]. By default, OpenSSL needlessly recalculates both Montgomery parameters R and $R^{-1} \bmod N$ on every decryption. Our minor patch allows OpenSSL to cache both values between decryptions with the same key. Our patch does not affect any other aspect of the RSA decryption other than caching these values. Figure 5 shows the results of an attack both with and without the patch.

The zero-one gap is shifted because the resulting code will have a different execution profile, as discussed in the previous experiment. While our specific patch decreases the size of the zero-one gap, other patches may increase the zero-one gap. This shows the danger of assuming a specific application is not vulnerable due to timing attack tests, as even a small patch can change the run-time profile and either increase or decrease the zero-one gap. Developers should instead rely upon proper algorithmic defenses as discussed in section 6.

5.6 Experiment 5 - Interprocess vs. Local Network Attacks

To show that local network timing attacks are practical, we connected two computers via a 10/100 Mb Hawking switch, and compared the results of the attack interprocess vs. inter-network. Figure 6 shows that the network does not seriously diminish the effectiveness of the attack. The noise from the network is eliminated by repeated sampling, giving a similar zero-one gap to inter-process. We note that in our tests a zero-one gap of approximately 1 millisecond is sufficient to receive a strong indicator, enabling a successful attack. Thus, networks with less than 1ms of variance are vulnerable.



(a) The zero-one gaps when attacking Apache+mod_SSL and stunnel separated by one switch.

(b) The zero-one gap when attacking Apache+mod_SSL separated by several routers and a network backbone.

Figure 7: Applications using OpenSSL 0.9.7 are vulnerable, even on a large network.

Inter-network attacks allow an attacker to also take advantage of faster CPU speeds for increasing the accuracy of timing measurements. Consider machine 1 with a slower CPU than machine 2. Then if machine 2 attacks machine 1, the faster clock cycle allows for finer grained measurements of the decryption time on machine 1. Finer grained measurements should result in fewer queries for the attacker, as the zero-one gap will be more distinct.

5.7 Experiment 6 - Attacking SSL Applications on the Local Network

We show that OpenSSL applications are vulnerable to our attack from the network. We compiled Apache 1.3.27 + mod_SSL 2.8.12 and stunnel 4.04 per the respective "INSTALL" files accompanying the software. Apache+mod_SSL is a commonly used secure web server. stunnel allows TCP/IP connections to be tunneled through SSL.

We begin by showing servers connected by a single switch are vulnerable to our attack. This scenario is relevant when the attacker has access to a machine near the OpenSSL-based server. Figure 7(a) shows the result of attacking stunnel and mod_SSL where the attacking client is separated by a single switch. For reference, we also include the results for a similar attack against the simple RSA decryption server from the previous experiments.

Interestingly, the zero-one gap is larger for Apache+mod_SSL than either the simple RSA decryption server or stunnel. As a result, successfully attacking Apache+mod_SSL requires fewer queries than stunnel. Both applications have a sufficiently large zero-one gap to be considered vulnerable.

To show our timing attacks can work on larger networks, we separated the attacking client from the Apache+mod_SSL server by our campus backbone. The webserver was hosted in a separate building about a half mile away, separated by three routers and a number of switches on the network backbone. Figure 7(b) shows the effectiveness of our attack against Apache+mod_SSL on this larger LAN, contrasted with our previous experiment where the attacking client and server are separated by only one switch.

This experiment highlights the difficulty in determining the minimum number of queries for a successful attack. Even though both stunnel and mod_SSL use the exact same OpenSSL libraries and use the same parameters for negotiating the SSL handshake, the run-time differences result in different zero-one gaps. More importantly, our attack works even when the attacking client and application are separated by a large network.

6 Defenses

We discuss three possible defenses. The most widely accepted defense against timing attacks is to perform RSA blinding. The RSA blinding operation calculates $x = r^e g \bmod N$ before decryption, where r is random, e is the RSA encryption exponent, and g is the ciphertext

to be decrypted. x is then decrypted as normal, followed by division by r, i.e. $x^e/r \bmod N$. Since r is random, x is random and timing the decryption should not reveal information about the key. Note that r should be a new random number for every decryption. According to [17] the performance penalty is 2% 10%, depending upon implementation. Netscape/Mozilla's NSS library uses blinding. Blinding is available in OpenSSL, but not enabled by default in versions prior to 0.9.7b. Figure 8 shows that blinding in OpenSSL 0.9.7b defeats our attack. We hope this paper demonstrates the necessity of enabling this defense.

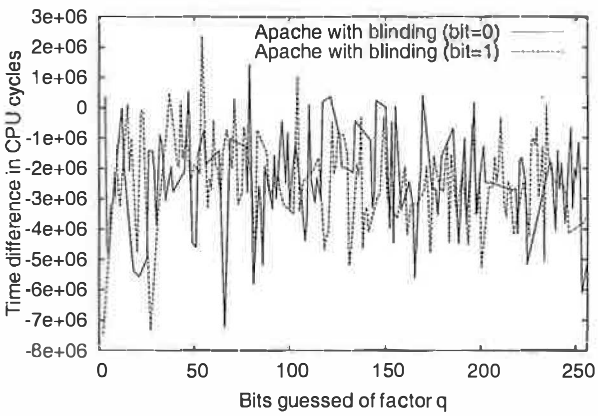


Figure 8: Our attack against Apache+mod_SSL using OpenSSL 0.9.7b is defeated because blinding is enabled by default.

Two other possible defenses are suggested often, but are a second choice to blinding. The first is to try and make all RSA decryptions not dependent upon the input ciphertext. In OpenSSL one would use only one multiplication routine and always carry out the extra reduction in Montgomery's algorithm, as proposed by Schindler in [18]. If an extra reduction is not needed, we carry out a "dummy" extra reduction and do not use the result. Karatsuba multiplication can always be used by calculating $c \bmod p_i * 2^m$, where c is the ciphertext, p_i is one of the RSA factors, and $m = \log_2 p_i - \log_2 (c \bmod p_i)$. After decryption, the result is divided by $2^{md} \bmod q$ to yield the plaintext. We believe it is harder to create and maintain code where the decryption time is not dependent upon the ciphertext. For example, since the result is never used from a dummy extra reduction during Montgomery reductions, it may inadvertently be optimized away by the compiler.

Another alternative is to require all RSA computations to be quantized, i.e. always take a multiple of some predefined time quantum. Matt Blaze's quantize library [1] is an example of this approach. Note that *all* decryptions must take the maximum time of *any* decryption, otherwise, timing information can still be used to leak information about the secret key.

Currently, the preferred method for protecting against timing attacks is to use RSA blinding. The immediate drawbacks to this solution is that a good source of randomness is needed to prevent attacks on the blinding factor, as well as the small performance degradation. In OpenSSL, neither drawback appears to be a significant problem.

7 Conclusion

We devised and implemented a timing attack against OpenSSL — a library commonly used in web servers and other SSL applications. Our experiments show that, counter to current belief, the timing attack is effective when carried out between machines separated by multiple routers. Similarly, the timing attack is effective between two processes on the same machine and two Virtual Machines on the same computer. As a result of this work, several crypto libraries, including OpenSSL, now implement blinding by default as described in the previous section.

8 Acknowledgments

This material is based upon work supported in part by the National Science Foundation under Grant No. 0121481 and the Packard Foundation. We thank the reviewers, Dr. Monica Lam, Ramesh Chandra, Constantine Sapuntzakis, Wei Dai, Art Manion and CERT/CC, and Dr. Werner Schindler for their comments while preparing this paper. We also thank Nelson Bolyard, Geoff Thorpe, Ben Laurie, Dr. Stephen Henson, Richard Levitte, and the rest of the OpenSSL, mod_SSL, and stunnel development teams for their help in preparing patches to enable and use RSA blinding.

References

[1] Matt Blaze. Quantize wrapper library. `http://islab.oregonstate.edu/documents/People/blaze`.

[2] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults. *Lecture Notes in Computer Science*, 1233:37–51, 1997.

[3] D. Coppersmith. Small solutions to polynomial equations, and low exponent RSA vulnerabilities. *Journal of Cryptology*, 10:233–260, 1997.

[4] Jean-Francois Dhem, Francois Koeune, Philippe-Alexandre Leroux, Patrick Mestre, Jean-Jacques Quisquater, and Jean-Louis Willems. A practical implementation of the timing attack. In *CARDIS*, pages 167–182, 1998.

[5] Peter Gutmann. Cryptlib. `http://www.cs.auckland.ac.nz/~pgut001/cryptlib/`.

[6] Intel. Vtune performance analyzer for linux v1.1. `http://www.intel.com/software/products/vtune`.

[7] Intel. Using the RDTSC instruction for performance monitoring. Technical report, 1997.

[8] Intel. Ia-32 intel architecture optimization reference manual. Technical Report 248966-008, 2003.

[9] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis: Leaking secrets. In *Crypto 99*, pages 388–397, 1999.

[10] Paul Kocher. Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems. *Advances in Cryptology*, pages 104–113, 1996.

[11] Alfred Menezes, Paul Oorschot, and Scott Vanstone. *Handbook of Applied Cryptography*. CRC Press, October 1996.

[12] mod_SSL Project. mod_ssl. `http://www.modssl.org`.

[13] Peter Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[14] GNU Project. libgcrypt. `http://www.gnu.org/directory/security/libgcrypt.html`.

[15] OpenSSL Project. Openssl. `http://www.openssl.org`.

[16] Rao, Josyula, Rohatgi, and Pankaj. Empowering side-channel attacks. Technical Report 2001/037, 2001.

[17] RSA Press Release. `http://www.otn.net/onthenet/rsaqa.htm`, 1995.

[18] Werner Schindler. A timing attack against RSA with the chinese remainder theorem. In *CHES 2000*, pages 109–124, 2000.

[19] Werner Schindler. A combined timing and power attack. *Lecture Notes in Computer Science*, 2274:263–279, 2002.

[20] Werner Schindler. Optimized timing attacks against public key cryptosystems. *Statistics and Decisions*, 20:191–210, 2002.

[21] Werner Schindler, Franois Koeune, and Jean-Jacques Quisquater. Improving divide and conquer attacks against cryptosystems by better error detection/correction strategies. *Lecture Notes in Computer Science*, 2260:245–267, 2001.

[22] Werner Schindler, Franois Koeune, and Jean-Jacques Quisquater. Unleashing the full power of timing attack. Technical Report CG-2001/3, 2001.

[23] stunnel Project. stunnel. `http://www.stunnel.org`.

802.11 Denial-of-Service Attacks: Real Vulnerabilities and Practical Solutions

John Bellardo and Stefan Savage
Department of Computer Science and Engineering
University of California at San Diego

Abstract

The convenience of 802.11-based wireless access networks has led to widespread deployment in the consumer, industrial and military sectors. However, this use is predicated on an implicit assumption of confidentiality and availability. While the security flaws in 802.11's basic confidentially mechanisms have been widely publicized, the threats to network availability are far less widely appreciated. In fact, it has been suggested that 802.11 is highly susceptible to malicious denial-of-service (DoS) attacks targeting its management and media access protocols. This paper provides an experimental analysis of such 802.11-specific attacks – their practicality, their efficacy and potential low-overhead implementation changes to mitigate the underlying vulnerabilities.

1 Introduction

The combination of free spectrum, efficient channel coding and cheap interface hardware have made 802.11-based access networks extremely popular. For a couple hundred dollars a user can buy an 802.11 access point that seamlessly extends their existing network connectivity for almost 100 meters. As a result, the market for 802.11-based LANs exceeded $1 Billion in 2001 and includes widespread use in the home, enterprise and government/military sectors, as well as an emerging market in public area wireless networks. However, this same widespread deployment makes 802.11-based networks an attractive target for potential attackers. Indeed, recent research has demonstrated basic flaws in 802.11's encryption mechanisms [FMS01, BGW01] and authentication protocols [ASJZ01] – ultimately leading to the creation of a series of protocol extensions and replacements (e.g., WPA, 802.11i, 802.1X) to address these problems. However, most of this work has focused *primarily* on the requirements of access control and confidentiality, rather than availability.

In contrast, this paper focuses on the threats posed by denial-of-service (DoS) attacks against 802.11's MAC protocol. Such attacks, which prevent legitimate users from accessing the network, are a vexing problem in all networks, but they are particularly threatening in the wireless context. Without a physical infrastructure, an attacker is afforded considerable flexibility in deciding where and when to attack, as well as enhanced anonymity due to the difficulty in locating the source of individual wireless transmissions. Moreover, the relative immaturity of 802.11-based network management tools makes it unlikely that a well-planned attack will be quickly diagnosed. Finally, as we will show, vulnerabilities in the 802.11 MAC protocol allow an attacker to selectively or completely disrupt service to the network using relatively few packets and low power consumption.

This paper makes four principal contributions. First, we provide a description of vulnerabilities in the 802.11 management and media access services that are vulnerable to attack. Second, we demonstrate that all such attacks are practical to implement by circumventing the normal operation of the firmware in commodity 802.11 devices. Third, we implement two important classes of denial-of-service attacks and investigate the range of their practical effectiveness. Finally, we describe, implement and evaluate non-cryptographic countermeasures that can be implemented in the firmware of existing MAC hardware.

The rest of this paper is structured as follows: Section 2 describes related security research conducted by others in academia, as well as unpublished, but contemporaneous, work from the "black-hat" security community. Section 3 describes and categorizes existing denial-of-service vulnerabilities in 802.11's MAC protocol. In Section 4 we use live experiments and simulation to analyze the practicality and efficacy of these attacks, followed by an evaluation of low-overhead countermeasures to mitigate

the underlying vulnerabilities. Finally, we summarize our findings in Section 5.

2 Related Work

A great deal of research has already been focused on 802.11 network security. Most of this work has focused on weaknesses in the wired equivalency protocol (WEP) intended to provide data privacy between 802.11 clients and access points. WEP relies on shared secret keys to support a challenge-response authentication protocol and for encrypting data packets. In 2001, Fluhrer et al. identified recurring weak keys in WEP, and showed how to use them to recover the secret key [FMS01]. Once the key is known, an attacker can both fully utilize network resources and monitor the traffic of other network nodes. In a recent paper, Stubblefield et al., demonstrate an implementation of this attack that was able to recover a 128-bit WEP key purely through passive monitoring [SIR02]. In addition, Borisov et al. have identified vulnerabilities that allow WEP-protected frames to be modified, new frames to be injected, authentication frames to be spoofed and plain text to be recovered from encrypted frames – all without knowing the shared secret key [BGW01].

While these works comprise the best known body of 802.11 security research, there has also been some attention focused on denial-of-service vulnerabilities unique to 802.11. As part of his PhD thesis, Lough identifies a number of security vulnerabilities in the 802.11 MAC protocol, including those that lead to the deauthentication/disassociation and virtual carrier-sense attacks presented in this paper [Lou01]. However, while Lough's thesis identifies these vulnerabilities, it does not validate them empirically. We demonstrate that such validation is critical to assessing the true threat of such attacks.

In addition to Lough's work, Faria and Cheriton consider the problems posed by authentication DoS attacks. They identify those assumption violations that lead to the vulnerabilities and propose a new authentication framework to address the problems [FC02]. Unlike their work, this paper focuses on validating the impact of the attacks and developing light-weight solutions that do not require significant changes to existing standards or extensive use of cryptography.

The deauthentication/disassociation attack is fairly straightforward to implement and while writing this paper we discovered several in the "black hat" community who had done so before us. Lacking publication dates it is difficult to determine the ordering of these efforts, but we are aware of three implementations to date: one by Baird and Lynn (AirJack) presented at BlackHat Briefings in July of 2002, another due to Schiffman and presented at the same event (Omerta), and a tool by Floeter (void11) that appears to be roughly contemporaenous [LB02, Sch02, Flo02]. As part of his implementation, Schiffman also discusses a general purpose toolkit, called Radiate, for injecting raw 802.11 frames into the channel. However, since this toolkit works through the firmware it is only able to generate a subset of legitimate 802.11 frames. Compared to this previous work, our contribution lies in evaluating the impact of the attack, providing a cheap means to mitigate such attacks and in providing an infrastructure for mounting a wider class of attacks (including the virtual carrier-sense attack).

Congestion-based MAC layer denial of service attacks have also been studied previously. Gupta et al. examined DoS attacks in 802.11 ad hoc networks and show that traditional wireline-based detection and prevention approaches do not work, and propose the use of MAC layer fairness to mitigate the problem [GKF02]. Kyasanur and Vaidya also look at congestion-based MAC DoS attacks, but from a general 802.11 prospective, not the purely ad hoc prospective [KV03]. They propose a straightforward method for detecting such attacks. In addition they propose and simulate a defense where uncompromised nodes cooperate to control the frame rate at the compromised node. Compared to these papers, we focus on attacks on the 802.11 MAC protocol itself rather than pure resource consumption attacks.

Finally, to provide a long-term solution to 802.11's security problems, the 802.11 TGi working group has proposed the standard use of the 802.1X protocol [IEE01] for authentication in future versions of 802.11 products, in addition to both short-term and long-term modifications to the privacy functions. However, while the working group is clearly aware of threats from unauthenticated management frames and spoofed control frames (e.g., [Abo02, Moo02]), to the best of our knowledge there is no protection against such attacks in the current drafts under discussion.

3 Vulnerabilities

The 802.11 MAC layer incorporates functionality uniquely designed to address problems specific to wireless networks. In particular, this includes the ability to discover networks, join and leave networks, and coordinate access to the radio medium.

The vulnerabilities discussed in this section result directly from this additional functionality and can be broadly placed into two categories: identity and media-access control.

3.1 Identity Vulnerabilities

Identity vulnerabilities arise from the implicit trust 802.11 networks place in a speaker's source address. As is the case with wired Ethernet hosts, 802.11 nodes are identified at the MAC layer with globally unique 12 byte addresses. A field in the MAC frame holds both the senders and the receivers addresses, as reported by the sender of the frame. For "class one" frames, including most management and control messages, standard 802.11 networks do not include any mechanism for verifying the correctness of the self-reported identity. Consequently, an attacker may "spoof" other nodes and request various MAC-layer services on their behalf. This leads to several distinct vulnerabilities.

3.1.1 Deauthentication

Exemplifying this problem is the *deauthentication attack*. After an 802.11 client has selected an access point to use for communication, it must first authenticate itself to the AP before further communication may commence. Moreover, part of the authentication framework is a message that allows clients and access points to explicitly request deauthentication from one another. Unfortunately, this message itself is not authenticated using any keying material. Consequently the attacker may spoof this message, either pretending to be the access point or the client, and direct it to the other party (see Figure 1). In response, the access point or client will exit the authenticated state and will refuse all further packets until authentication is reestablished. How long reestablishment takes is a function of how aggressively the client will attempt to reauthenticate and any higher-level timeouts or backoffs that may suppress the demand for communication. By repeating the attack persistently a client may be kept from transmitting or receiving data indefinitely.

One of the strengths of this attack is its great flexibility: an attacker may elect to deny access to individual clients, or even rate limit their access, in addition to simply denying service to the entire channel. However, accomplishing these goals efficiently requires the attacker to promiscuously monitor the channel and send deauthentication messages only when a new authentication has successfully taken place (indicated by the client's attempt to associate with the access point). As well, to prevent a client from "escaping" to a neighboring access point, the attacker must periodically scan all channels to ensure that the client has not switched to another overlapping access point.

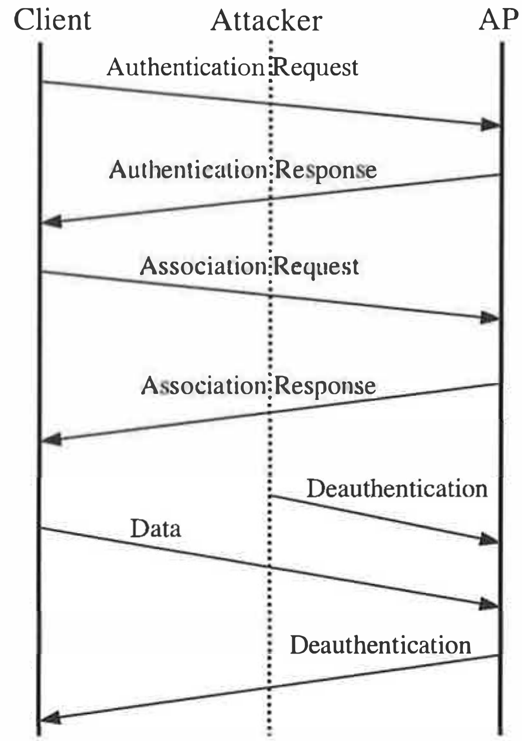


Figure 1: Graphical depiction of the deauthentication attack. Note that the attacker needs only generate one packet for every six exchanged between the client and access point.

3.1.2 Disassociation

A very similar vulnerability may be found in the association protocol that follows authentication. Since a client may be authenticated with multiple access points at once, the 802.11 standard provides a special association message to allow the client and access point to agree which access point shall have responsibility for forwarding packets to and from the wired network on the client's behalf. As with authentication, association frames are unauthenticated, and 802.11 provides a disassociation message similar to the deauthentication message described earlier. Exploiting this vulnerability is functionally identical to the deauthentication attack. However, it is worth noting that the disassociation attack is slightly less efficient than the deauthentication attack. This is because deauthentication forces the victim node to do more work to return to the as-

sociated state than does disassociation, ultimately requiring less work on the part of the attacker.

3.1.3 Power Saving

The power conservation functions of 802.11 also present several identity-based vulnerabilities. To conserve energy, clients are allowed to enter a sleep state during which they are unable to transmit or receive. Before entering the sleep state the client announces its intention so the access point can start buffering any inbound traffic for the node. Occasionally the client awakens and polls the access point for any pending traffic. If there is any buffered data at this time, the access point delivers it and subsequently discards the contents of its buffer. By spoofing the polling message on behalf of the client, an attacker cay cause the access point to discard the clients packets while it is asleep.

Along the same vein, it is potentially possible to trick the client node into thinking there are no buffered packets at the access point when in fact there are. The presence of buffered packets is indicated in a periodically broadcast packet called the traffic indication map, or TIM. If the TIM message itself is spoofed, an attacker may convince a client that there is no pending data for it and the client will immediately revert back to the sleep state.

Finally, the power conservation mechanisms rely on time synchronization between the access point and its clients so clients know when to awake. Key synchronization information, such as the period of TIM packets and a timestamp broadcast by the access point, are sent unauthenticated and in the clear. By forging these management packets, an attacker can cause a client node to fall out of sync with the access point and fail to wake up at the appropriate times.

While all of the vulnerabilities in this section could be resolved with appropriate authentication of all messages, it seems unlikely that such a capability will emerge soon. With an installed base of over 15 million legacy 802.11 devices, the enormous growth of the public-area wireless access market and the managerial burden imposed by the shared key management of 802.1X, it seems unlikely that there will be universal deployment of mutual authentication infrastructure any time soon. Moreover, it is not clear whether future versions of the 802.11 specification will protect management frames such as deauthentication (while it is clear they are aware of the problem, the current work of the TGi working group still leaves the deauthentication operation unprotected).

3.2 Media Access Vulnerabilities

802.11 networks go through significant effort to avoid transmit collisions. Due to hidden terminals perfect collision detection is not possible and a combination of physical carrier-sense and virtual carrier-sense mechanisms are employed in tandem to control access to the channel [BDSZ94]. Both of these mechanisms may be exploited by an attacker.

First, to prioritize access to the radio medium four time windows are defined. For the purposes of this discussion only two are important: the *Short Interframe Space* (SIFS) and the longer *Distributed Coordination Function Interframe Space* (DIFS). Before *any* frame can be sent the sending radio must observe a quiet medium for one of the defined window periods. The SIFS window is used for frames sent as part of a preexisting frame exchange (for example, the explicit ACK frame sent in response to a previously transmitted data frame). The DIFS window is used for nodes wishing to initiate a new frame exchange. To avoid all nodes transmitting immediately after the DIFS expires, the time after the DIFS is subdivided into slots. Each transmitting node randomly and with equal probability picks a slot in which to start transmitting. If a collision does occur (indicated implicitly by the lack of an immediate acknowledgment), the sender uses a random exponential backoff algorithm before retransmitting.

Since every transmitting node must wait at least an SIFS interval, if not longer, an attacker may completely monopolize the channel by sending a short signal before the end of every SIFS period. While this attack would likely be highly effective, it also requires the attacker to expend considerable energy. A SIFS period is only 20 microseconds on 802.11b networks, leading to a duty cycle of 50,000 packets per second in order to disable all access to the network.

A more serious vulnerability arises from the virtual carrier-sense mechanism used to mitigate collisions from hidden terminals. Each 802.11 frame carries a *Duration* field that indicates the number of microseconds that the channel is reserved. This value, in turn, is used to program the *Network Allocation Vector* (NAV) on each node. Only when a node's NAV reaches 0 is it allowed to transmit. This feature is principally used by the explicit *request to send* (RTS) / *clear to send* (CTS) handshake that can be used to synchronize access to the channel when a hidden terminal may be interfering with transmissions.

During this handshake the sending node first sends a small RTS frame that includes a duration

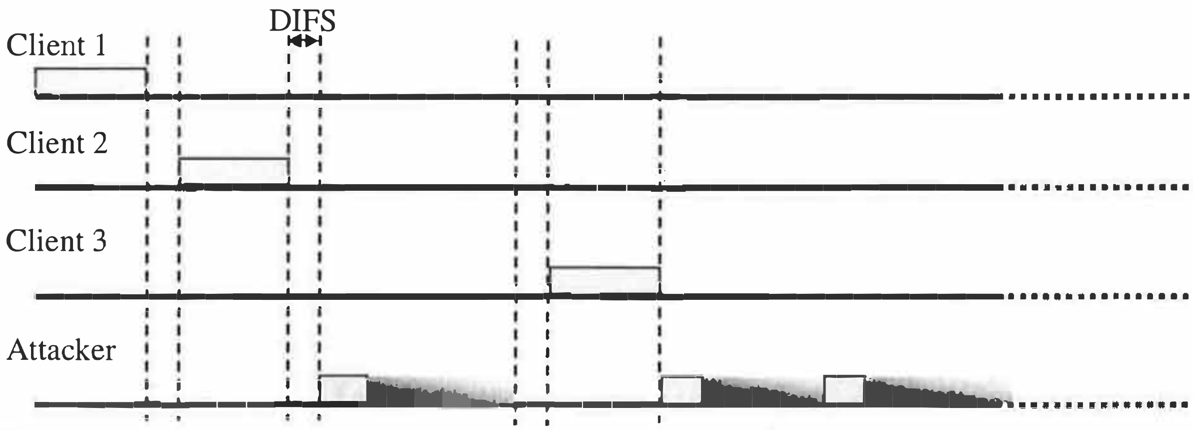


Figure 2: Graphical depiction of the virtual carrier-sense attack in action. The gradient portion of the attacker's frame indicates time reserved by the duration field although no data is actually sent. Continually sending the attack frames back to back prevents other nodes from sending legitimate frames.

large enough to complete the RTS/CTS sequence – including the CTS frame, the data frame, and the subsequent acknowledgment frame. The destination node replies to the RTS with a CTS, containing a new duration field updated to account for the time already elapsed during the sequence. After the CTS is sent, every node in radio range of either the sending or receiving node will have updated their NAV and will defer all transmissions for the duration of the future transaction. While the RTS/CTS feature is rarely used in practice, respecting the virtual carrier-sense function indicated by the duration field is mandatory in all 802.11 implementations.

An attacker may exploit this feature by asserting a large duration field, thereby preventing well-behaved clients from gaining access to the channel (as shown in Figure 2). While it is possible to use almost any frame type to control the NAV, including an ACK, using the RTS has some advantages. Since a well-behaved node will always respond to RTS with a CTS, an attacker may co-opt legitimate nodes to propagate the attack further than it could on its own. Moreover, this approach allows an attacker to transmit with extremely low power or using a directional antennae, thereby reducing the probability of being located.

The maximum value for the NAV is 32767, or roughly 32 milliseconds on 802.11b networks, so in principal an attacker need only transmit approximately 30 times a second to jam all access to the channel. Finally, it is worth noting that RTS, CTS and ACK frames are not authenticated in any current or upcoming 802.11 standard. However, even if they were authenticated, this would only provide non-repudiation since, by design, the virtual-carrier sense feature impacts all nodes on the same channel.

4 Practical Attacks and Defenses

While the previous vulnerabilities are severe in principal, understanding their true threat potential requires evaluating the practicality of implementing them and how well they perform in practice. In this section we examine these issues as well as discussing the efficacy of several low-overhead defense mechanisms.

4.1 802.11 Attack Infrastructure

From a purely practical perspective, a key engineering question is, "Can an attack be generated with commodity hardware?" While theoretical vulnerabilities are clearly important, we feel that attacks with software implementations represent a qualitatively greater threat since they are available to a dramatically expanded set of potential attackers.

At first glance this appears to be a trivial problem since all 802.11 Network Interface Cards (NIC) are inherently able to generate arbitrary frames. However, in practice, all 802.11(a,b) devices we are aware of implement key MAC functions in firmware and moderate access to the radio through a constrained interface. The implementation of this firmware, in turn, dictates the limits of how a NIC can be used by an attacker. Indeed, in reviewing preprints of this paper, several 802.11 experts declared the virtual carrier-sense attack infeasible in practice due to such limitations.

In testing a wide variety of 802.11 NICs we have found that most allow the generation of management frames necessary to exploit the identity attacks described earlier – typically using semi-documented or undocumented modes of operation, such as HostAP

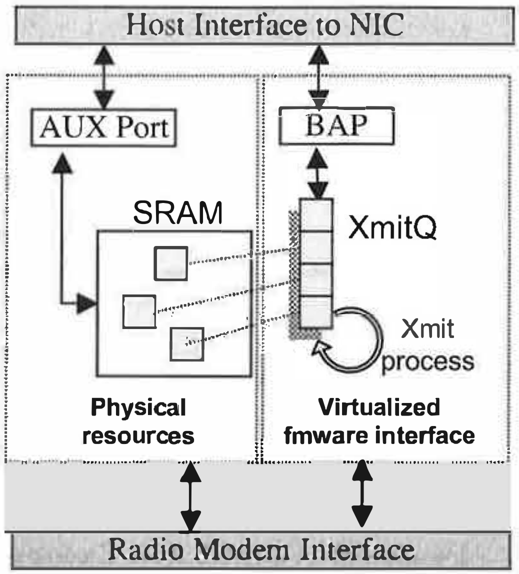


Figure 3: A block diagram depicting how the "aux port" can be used to circumvent the limitations imposed by the firmware. By using this raw memory interface, the host can transform "normal" packets into arbitrary 802.11 frames as they are transmitted.

and HostBSS mode in Intersil firmware. However, these same devices do **not** typically allow the generation of any control frames, permit other key fields (such as Duration and FCS) to be specified by the host, or allow reserved or illegal field values to be transmitted. Instead, the firmware overwrites these fields with appropriate values after the host requests that queued data be transmitted. While it might be possible to reverse-engineer the firmware to remove this limitation, we believe the effort to do so would be considerable. Instead, we have developed an alternative mechanism to sidestep the limitations imposed by the firmware interface. To understand our approach it is first necessary to understand the architecture of existing 802.11 products.

Most commodity 802.11 devices, including those using Intersil Prism, Lucent/Agere/Orinoco/Proxim Hermes and Cisco Aironet chipsets are based on an initial MAC design originated by Choice Microsystems (since acquired by Intersil). In this architecture, all low-level functions – including frame transmission, scheduling, acknowledgement, and fragmentation – are implemented in firmware while the host is simply responsible for managing data transfer to and from the device. Data transfer is achieved through a firmware-implemented "Buffer Access Path" (BAP) that shields the driver writer from the details of NIC memory management and synchronization. While the BAP interface will typically accept raw 802.11 frames, these packets are then further interpreted by concurrent firmware processes. As a result, only a subset of potential frames can be successfully transmitted by the host.

However, Choice-based MACs also provide an unbuffered, unsychronized raw memory access interface for debug purposes – typically called the "aux port". By properly configuring the host and NIC, it is possible to write a frame via the BAP interface, locate it in the NIC's SRAM, request a transmission, and then modify the packet via the aux port – after the firmware has processed it, but before it is actually transmitted. This process is depicted in Figure 3. To synchronize the host and NIC, a simple barrier can be implemented by spinning on an 802.11 header field (such as duration) that is overwritten by the firmware. Alternatively, the host can continuously overwrite if synchronization is unnecessary. In practice, this "data race" approach, while undeniably ugly, is both reliable and permits the generation of arbitrary 802.11 MAC frames. Using this method we are able to implement any of the attacks previously described using off-the-shelf hardware. We believe we are the first to demonstrate this capability using commodity equipment.

Our prototype, called *Swat*, consists of an iPAQ H3600 Pocket PC, running Familiar Linux, with a DLink DWL-650 PCMCIA 802.11 interface mounted in a standard PC Card sleeve. The entire device weighs approximately 375g (a bit over 12 oz) and is easily concealed in a coat pocket. More modern Pocket PCs, such as the Toshiba e740/e750 and the HP iPAQ 5450, include integral 802.11 functionally and could accomplish the same feats with roughly half the size and weight.

To experiment with denial-of-service attacks we have built a demonstration application that passively monitors wireless channels for APs and clients. Individual clients are identified initially by their MAC address, but as they generate traffic, a custom DNS resolver and a slightly modified version of dsniff [Son] is used to isolate better identifiers (e.g., userids, DNS address of IMAP server, etc). These identifiers can be used to select individual hosts for attack, or all hosts may be attacked en masse. The application and the actual device are pictured in Figure 4.

In the remainder of this section, we analyze the impact of the deauthentication attack and a preliminary defense mechanism, followed by a similar examination of the virtual carrier-sense attack and defense.

Figure 4: iPAQ H3600 with Dlink DWL-650 card, running Swat attack testing tool. Individual clients and AP's are identified either using MAC address or by passively monitoring the channel and extracting destination IP addresses and DNS names.

4.2 Deauthentication Attack

Our implementation of this attack promiscuously monitors all network activity, including non-data 802.11 frames, and matches the source and destination MAC address against a list of attack targets. If a data or association response frame is received from a target, we issue a spoofed deauthentication frame to the access point on behalf of the client. To avoid buffer overflow in congested networks on the attacking machine, deauthentication frames are rate limited to 10 frames per second per client. This limit is reset when an access point acknowledges receipt of a deauthentication frame.

We tested this implementation in a small 802.11 network composed of 7 machines: 1 attacker, 1 access point, 1 monitoring station, and 4 legitimate clients. The access point was built using the Linux HostAP driver, which provides an in-kernel software-based access point. Each of the clients attempted to transfer, via ftp, a large file through the access point machine – a transfer which exceeded the testing period. We mounted two attacks on the network. The first, illustrated by the thin rectangle in Figure 5, was directed against a single client running MacOS X. This client's transfer was immediately halted, and even though the attack lasted less than ten seconds, the client did not resume transmitting at its previous rate for more than a minute. This amplification was due to a combination of an extended delay while the client probed for other access points and the exponential backoff being employed by the ftp server's TCP implementation.

The second attack, delineated by the wider rectangle in the same figure, was directed against all four clients. Service is virtually halted during this period, although the Windows XP client is able to send a number of packets successfully. This anomaly has two sources. First, these are not data packets from the ftp session but rather UDP packets used by Window's DCE RPC service and not subject to TCP's congestion control procedure. Second, there is a small race condition in our attack implementation between the time a client receives the successful association response and the time the attacker sends the deauthentication frame. The WinXP client used this small window to send approximately ten UDP packets before the attacking node shut them down. Modifying the implementation to send the deauthentication packets after both authentication and association would mitigate this effect.

A number of smaller, directed attacks were performed in addition to those in Figure 5. The small tests were done using the extended 802.11 infrastructure found at UCSD with varied victims. Recent versions of Windows, Linux, and the MacOS all gave up on the targeted access point and kept trying to find others. Slightly older versions of the same systems never attempted to switch access points and were completely disconnected using the less sophisticated version of the attack. The attack even caused one device, an HP Jornada Pocket PC, to consistently crash.

The deauthentication vulnerability can be solved directly by explicitly authenticating management frames and dropping invalid requests. However, the standardization of such capabilities is still some ways off and it is clear that legacy MAC designs do not have sufficient CPU capacity to implement this functionality as a software upgrade. Therefore, system-level defenses with low-overhead can still offer significant value. In particular, by delaying the effects of deauthentication or disassociation requests (e.g., by queuing such requests for 5-10 seconds) an AP has the opportunity to observe subsequent packets from the client. If a data packet arrives after a deauthentication or disassociation re-

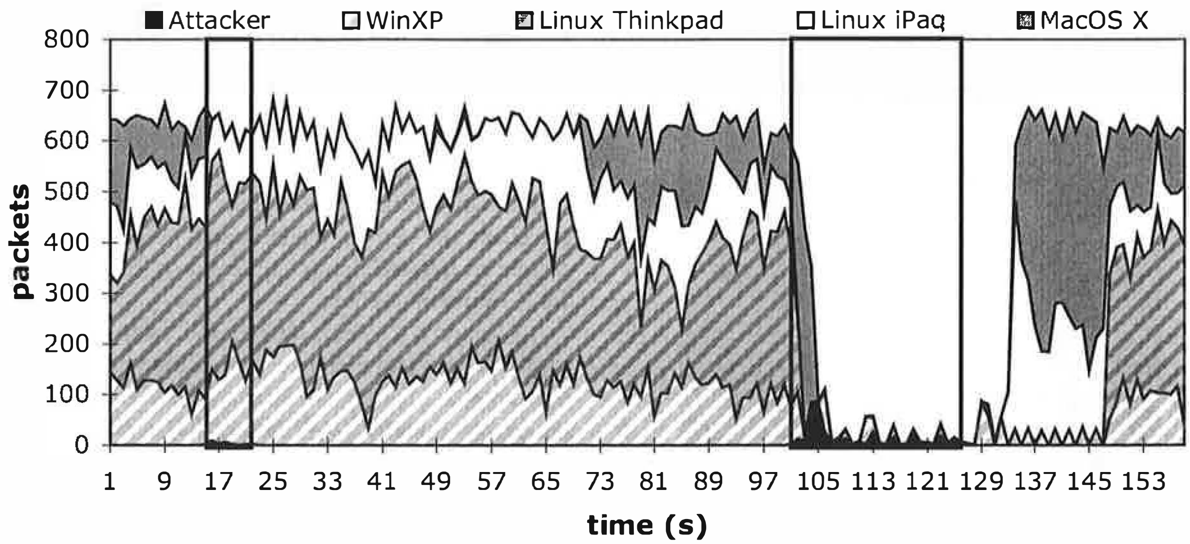


Figure 5: Packets sent by each of the 4 client nodes during the deauthentication attack. The first attack, against the MacOS client, started at second 15 and lasted 8 seconds. The second attack against all the clients started at 101 and lasted for 26 seconds. The attacking node consumes a negligible amount of bandwidth due to the rate limiting.

quest is queued, that request is discarded – since a legitimate client would never generate packets in that order. The same approach can be used in reverse to mitigate forged deauthentication packets sent to the client on behalf of the AP. This approach has the advantage that it can be implemented with a simple firmware modification to existing NICs and access points, without requiring a new management structure.

To test this defense we modified the access point used in our experiments as described above, using a timeout value of 10 seconds for each management request. We then executed the previous experiment again using the "hardened" access point. The equivalent results can be seen in Figure 6. From this graph it is difficult to tell that the attack is active, and the client nodes continue their activity oblivious to the misdirection being sent to the access point.

However, our proposed solution is not without drawbacks. In particular, it opens up a new vulnerability at the moment in which mobile clients roam between access points. The association message is used to determine which AP should receive packets destined for the mobile client. In certain circumstances leaving the old association established for an additional period of time may prevent the routing updates necessary to deliver packets through the new access point. Or, in the case of an adversary, the association could be kept open indefinitely by spoofing packets from the mobile client to the spoofed AP – keeping the association current. While both these situations are possible, we will argue that they are unlikely to represent a new threat in practice.

There are two main infrastructure configurations that support roaming. For lack of a better name we refer to these as "intelligent" and "dumb". In the "intelligent" configuration the access points have an explicit means of coordination. This coordination can be used to, among other things, update routes for and transfer buffered packets between access points when a mobile node changes associations. Since there is not currently a standard for this coordination function, AP's offering such capabilities typically use proprietary protocols that work only between homogenous devices. In contrast "dumb" access points have no explicit means of coordination and instead rely on the underlying layer-two distribution network (typically Ethernet) to reroute packets as a mobile client's MAC address appears at a new AP (and hence a new Ethernet switch port).

Intelligent infrastructures, due to their preexisting coordination, are easily modified to avoid the aforementioned problems caused by the deassociation timeout. Since the mobile node must associate with the new access point before it can transmit data, and since the access points are coordinated (either directly or through a third party), the old access point can be informed when the mobile node makes a new association. Based on this information the old access point can immediately honor the clients deauthentication request. While an attacker can spoof packets from the mobile host to create

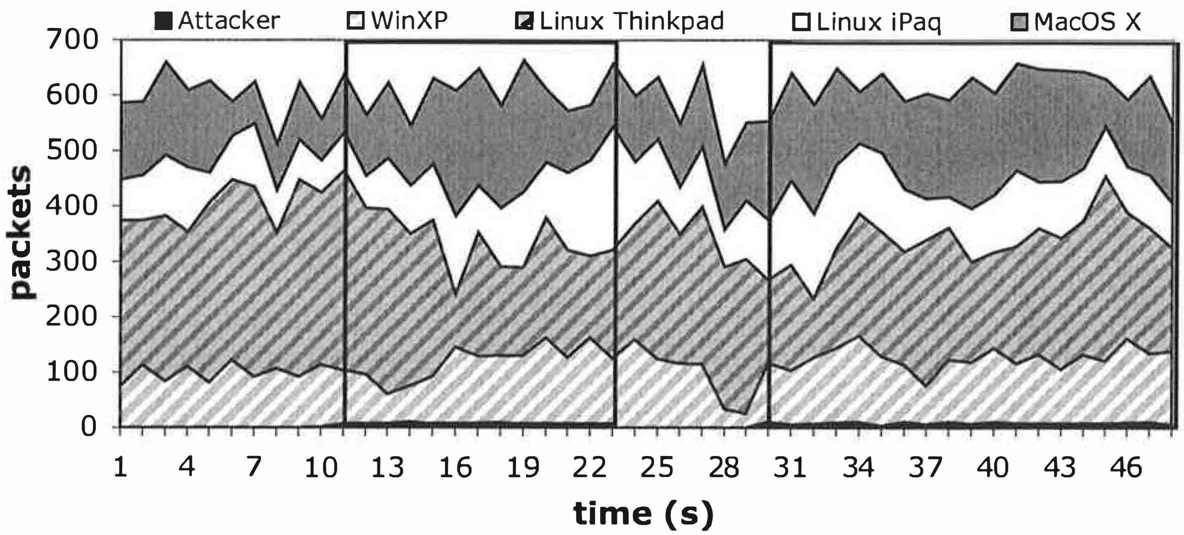


Figure 6: Packets sent by each of the 4 client nodes during the deauthentication attack with an access point modified to defend against this attack. The first attack, against the MacOS client, started at second 10 and lasted 12 seconds. The second attack against all the clients started at 30 and lasted through the end of the trace. The attacking node consumes a negligible amount of bandwidth due to the rate limiting.

confusion, this vulnerability exists without the addition of the deferred deassociation mechanisms we have described.

Dumb infrastructures are slightly more problematic because of their lack of coordination and reliance on the underlying network topology. If that underlying topology is a broadcast medium, which is a rarity these days, there is no problem because all packets are already delivered to all access points. If the underlying topology is switched, then a protocol is used (typically a spanning tree distribution protocol) to distribute which MAC addresses are served by which ports. Existing switches already gracefully support moving a MAC address from one port to another, but have problems when one MAC address is present across multiple ports. In the non-adversarial case the mobile node will switch access points, proceed to send data using the new access point, and cease sending data through the old access point. From the switches perspective this is equivalent to a MAC switching ports. The mobile node may not receive data packets until it has sent one – allowing the switch to learn its new port – but that limitation applies regardless of the deauthentication timeout. In the adversarial case the attacking node could generate spoofed traffic designed to confuse the switch. However, this does not represent a significant new vulnerability – even without the delay on deauthentication/disassociation an attacker can spoof a packet from an mobile client in order to create this conflict (including a WEP protected packet after key recovery).

4.3 Virtual carrier-sense attack

Motivated by the success of the previous attack, we built an implementation exploiting the NAV vulnerability. We generated a variety of packet streams with a range of large duration values – including continuous runs of RTS frames, CTS frames, and ACK frames destined for APs, hosts and unallocated addresses. We verified that packets were being sent as expected using a separate machine to monitor the channel being targeted. To our surprise, while our implementation carried out the attacks faithfully, they did not have the expected impact. We repeated these experiments using both Lucent WavePoint II and Apple Airport Extreme access points and with a variety of host NIC cards, all with the same results. After careful examination of traces collected during these attacks we have come to the conclusion that most of the devices available to us do not properly implement the 802.11 MAC specification and are improperly resetting their NAV. In particular, we have witnessed APs and NICs alike emit packets within a millisecond after the broadcast of a CTS frame with a duration of 32767. Figure 7 shows a trace excerpt illustrating this behavior – the initial CTS frame should keep the channel idle for 32ms, and yet after scarcely a millisecond has passed the channel is in use by another host. Such activity should be impossible under the 802.11 standard since nodes

Time	Src	Dest	Duration (ms)	Type
1.294020		:00:15:01	32767	802.11 CTS
1.295192	.10.2	.1.2	258	TCP Data
1.296540		:ea:e7:0f	0	802.11 Ack
1.297869	.1.2	.10.2	258	TCP Data
1.299084		:ea:e7:0f	0	802.11 Ack
1.300275	.1.2	.10.2	258	TCP Data
1.300439		:ea:e7:0f	0	802.11 Ack
1.302538		:00:15:01	32767	802.11 CTS
1.306110		:00:15:01	32767	802.11 CTS
1.309543	.10.2	.1.2	258	TCP Ack
1.309810		:ea:e7:0f	0	802.11 Ack
1.312237	.1.2	.10.2	258	TCP Data
1.313452		:ea:e7:0f	0	802.11 Ack

Figure 7: Excerpt from a typical virtual carrer-sense attack trace using CTS frames. The addresses have been shortened for brevity. MAC address :ea:e7:0f is the access point, and :00:15:01 is an unallocated MAC address. .1.2 is the uploading client, and .10.2 is the receiving machine. The first TCP data frame is sent 1.1 ms after a CTS that reserved the medium for > 32 ms. In the second CTS sequence the data frame is sent after 3.4 ms.

receiving the CTS cannot assume that they will be able to sense the carrier (or even significant radio energy) since the transmitter may be a hidden terminal. We have not conducted a thorough survey of 802.11 gear, so these deficiencies may be unique to the hardware in our environment. However, given the prevalence of the Choice design we would not be surprised if this bug is prevalent.

Under the assumption that these bugs will be removed in future 802.11 products (since they effectively prevent RTS/CTS from working as well as the 802.11 Point Coordinator Function and all related Quality-of-Service services based on 802.11) the remainder of this section explores the NAV vulnerability in the context of the popular ns simulator (which implements the protocol faithfully).

We implemented the virtual carrier-sense attack by modifying the ns [NS] 802.11 MAC layer implementation to allow arbitrary duration values to be sent periodically, 30 times a second, by the attacker. The attacker's frames were sent using the normal 802.11 access timing restrictions, which was necessary to prevent the attacker from excessively colliding with other in-flight frames (and thereby increase the amount of work required of the attacker). In addition the attacker was modified to ignore all duration values transmitted from any other node. The network topology was chosen to mimic many existing 802.11 infrastructure deployments: a single access point node, through who all traffic was being sent, 18 static client nodes and 1 static attacker node, all within radio distance of the access point. As with the previous experiments, ftp was used to generate the long-lived network traffic. We simulated attacks using ACK frames with large duration values, as well as the RTS/CTS sequence described earlier. Figure 8 shows the ACK flavor of the virtual carrier-sense attack in action, but both provided similar results: the channel is completely blocked for the duration of the attack.

The virtual carrier-sense attack is much harder to defend against in practice than the deauthentication attack.

One approach to mitigate its effects is to place a limit on the duration values accepted by nodes. Any packet containing a larger duration value is simply truncated to the maximum value allowable. Strict adherence to the required use of the NAV feature indicates two different limits: a low cap and a high cap. The low cap has a value equal to the amount of time required to send an ACK frame, plus media access backoffs for that frame. The low cap is usable when the only packet that can follow the observed packet is an ACK or CTS. This includes RTS and all management (association, etc) frames. The high cap, on the other hand, is used when it is valid for a data packet to follow the observed frame. The limit in this case needs to include the time required to send the largest data frame, plus the media access backoffs for that frame. The high cap must be used in two places: when observing an ACK (because the ACK my be part of a MAC level fragmented packet) and when observing a CTS.

We modified our simulation to add these limits, assuming that a value of 1500 bytes as the largest packet. While this is not strictly the largest packet that can be sent in an 802.11 network, it is the largest packet sent in practice because 802.11 networks are typically bridged to Ethernet, which has a roughly 1500 byte MTU. Figure 9 shows a simulation of this defense under the same conditions as the prior simulation. While there is still significant perturbation, many of the individual sessions are able to make successful forward progress. However, we found that simply by increasing the attacker's frequency to 90 packets per second, the network could still be shut down. This occurs because the attacker is using ACK frames, whose impact on the NAV is limited by the high cap.

To further improve upon this result requires us to abandon portions of the standard 802.11 MAC functionality. At issue is the inherent trust that nodes place in the duration value sent by other nodes. By considering the different frame types that

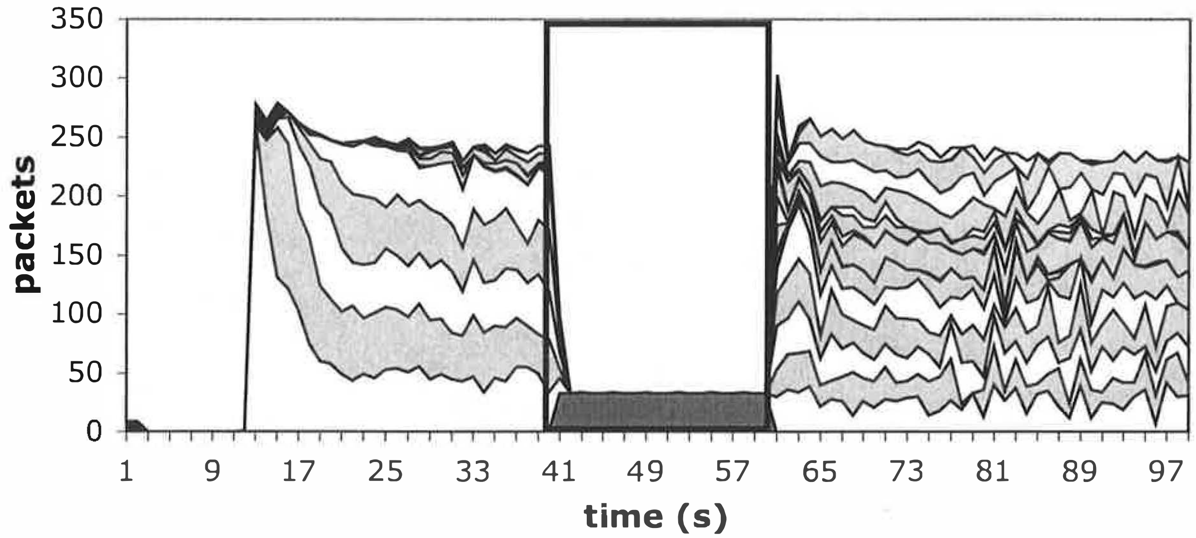


Figure 8: Results from the ACK based virtual carrier-sense attack simulation with 18 client nodes. The attack begins at time 40 and ends at time 60. The dark region at the bottom of the graph during the attack is the attacker.

carry duration values we can define a new interpretation of the duration that allows us to avoid most possible DoS attacks. The four key frame types that contain duration values are ACK, data, RTS, and CTS, and we consider each in turn.

Under normal circumstances the only time a ACK frame should carry a large duration value is when the ACK is part of a fragmented packet sequence. In this case the ACK is reserving the medium for the next fragment. If fragmentation is not used then there is no reason to respect the duration value contained in ACK frames. Since fragmentation is almost never used (largely due to the fact that default fragmentation thresholds significantly exceed the Ethernet MTU) removing it from operation altogether will have minimal impact on existing networks.

Like the ACK frame, the only legitimate occasion a data frame can carry a large duration value is if it is a subframe in a fragmented packet exchange. Since we have removed fragmentation from the network, we can safely ignore the duration values in all data frames.

The third frame type to be concerned with is the RTS frame. The RTS frame is only valid in an RTS-CTS-data transmission sequence. If an RTS is seen on the network, it follows that the node seeing the RTS will also be able to observe the data frame. The 802.11 specification precisely defines the time a CTS frame, and subsequent data frame, will be sent. Therefore the duration value in the RTS packet can be treated speculatively – respected until such time as a data frame should be sent. If the data frame is not observed at the correct time, either the sender has moved out of range or the RTS request was spoofed. In either case it is safe for the other node to undo the impact of this duration on the NAV. This interpretation is, in fact, allowed under the existing 802.11 standards.

The last frame to consider is the CTS frame. If a lone CTS frame is observed there are two possibilities: the CTS frame was unsolicited or the observing node is a hidden terminal. These are the *only* two cases possible, since if the observing node was not a hidden terminal it would have heard the original RTS frame and it would be handled accordingly. If the unsolicited CTS is addressed to a valid, in-range node, then only the valid node knows the CTS is bogus. It can prevent this attack by responding to such a CTS with a null function packet containing a zero duration value – effectively undoing the attackers channel reservation. However, if an unsolicited CTS is addressed to a nonexistent node, or a node out of radio range, this is indistinguishable from a legitimate hidden terminal. In this case, there is insufficient information for a legitimate node to act. The node issuing the CTS could be an attacker, or they may simply be responding to a legitimate RTS request that is beyond the radio range of the observer.

An imperfect approach to this final situation, is to allow each node to independently choose to ignore lone CTS packets as the fraction of time stalled on such requests increases. Since hidden terminals are

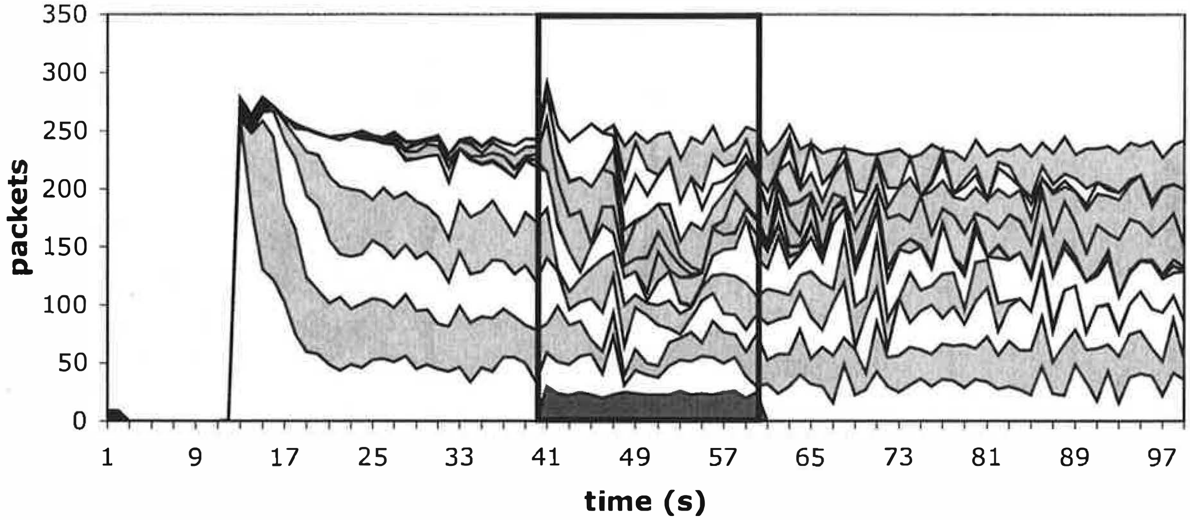


Figure 9: Results from the ACK based virtual carrier-sense attack simulation with 18 client nodes modified to implement defense. The attack begins at time 40 and ends at time 60. The dark region at the bottom of the graph during the attack is the attacker.

a not a significant efficiency problem in most networks (as evidenced by the fact that RTS/CTS are rarely employed and since the underlying functionality does not seem to work in many implementations) setting this threshold at 30 percent, will provide normal operation in most legitimate environments, but will prevent an attacker from claiming more than a third of the bandwidth using this attack.

It should also be noted that existing 802.11 implementations use different receive and carrier-sense thresholds. The different values are such that, in an open area, the interference radius of a node is approximately double its transmit radius. In the hidden terminal case this means that although the hidden terminal can not receive the data being transmitted, it still detects a busy medium and will not generate any traffic that would interfere with the data, so the possibility of an unsolicited CTS followed by an undetectable data packet is very low.

But ultimately the only foolproof solution to this problem is to extend explicit authentication to 802.11 control packets. Each client-generated CTS packet contains an implicit claim that it was sent in response to a legitimate RTS generated by an access point. However, to prove this claim, the CTS frame must contain a fresh and cryptographically signed copy of the originating RTS. If every client shares keying material with all surrounding access points it is then possible to authenticate lone CTS requests directly. However, such a modification is a significant alternation to the existing 802.11 standard, and it is unclear if it offers sufficient benefits relative to its costs. In the meantime, the system-level defenses we have described provide reasonable degrees of protection with extremely low implementation overhead and no management burden. Should media-access based denial-of-service attacks become prevalent, these solutions could be deployed quickly with little effort.

5 Conclusion

802.11-based networks have seen widespread deployment across many fields, mainly due to the physical conveniences of radio-based communication. This deployment, however, was predicated in part on the user expectation of confidentiality and availability. This paper addressed the availability aspect of that equation. We examined the 802.11 MAC layer and identified a number of vulnerabilities that could be exploited to deny service to legitimate users. We described software infrastructure for generating arbitrary 802.11 frames using commodity hardware and then used this platform to implement versions of the deauthentication and virtual carrier-sense attacks. We found that the former attack was highly effective in practice, while the latter is only a theoretical vulnerability due to implementation deficiencies in commodity 802.11 gear. In addition to demonstrating the attacks, we described and analyzed potential countermeasures. These countermeasures represent a stopgap measure, one that can be implemented with low overhead on existing hard-

ware, but not a long term substitute for appropriate per-packet authentication mechanisms. Overall, we believe this paper helps to underscore the care that must be taken when deploying 802.11 networks in mission critical applications.

6 Acknowledgements

We would like to thank the anonymous reviewers for their comments and in particular their pointers to the "blackhat" literature. Our shepherd, David Wagner, similarly provided input that was very helpful. We would like to thank Geoffrey Voelker, Anand Balachandran, and Daniel Faria for providing feedback on earlier drafts of this paper. Finally, a special thanks goes to the residents of csl-south at UCSD, who were at times unwitting victims of this research. This work was funded by DARPA Grant N66001-01-1-8933 and NIST Grant 60NANB1D0118.

References

[Abo02] Bernard Aboba. IEEE 802.1X Pre-Authentication. Presentation to 802.11 WGi, July 2002.

[ASJZ01] W.A. Arbaugh, N. Shankar, J.Wang, and K. Zhang. Your 802.11 Network has No Clothes. In *First IEEE International Conference on Wireless LANs and Home Networks*, Suntec City, Singapore, December 2001.

[BDSZ94] Vaduvur Bharghavan, Alan J. Demers, Scott Shenker, and Lixia Zhang. MACAW: A Media Access Protocol for Wireless LAN's. In *Proceedings of the ACM SIGCOMM Conference*, London, UK, September 1994.

[BGW01] Nikita Borisov, Ian Goldberg, and David Wagner. Intercepting Mobile Communications: The Insecurity of 802.11. In *Seventh Annual International Conference on Mobile Computing And Networking*, Rome, Italy, July 2001.

[FC02] Daniel B. Faria and David R. Cheriton. DoS and Authentication in Wireless Public Access Networks. In *Proceedings of the First ACM Workshop on Wireless Security (WiSe'02)*, September 2002.

[Flo02] Reyk Floeter. Wireless Lan Security Framework: void11. `http://www.wlsec.net/void11/`, 2002.

[FMS01] Scott Fluhrer, Itsik Mantin, and Adi Shamir. Weaknesses in the Key Scheduling Algorithm of RC4. *Lecture Notes in Computer Science*, 2259, 2001.

[GKF02] Vikram Gupta, Srikanth Krishnamurthy, and Michalis Faloutsos. Denial of Service Attacks at the MAC Layer in Wireless Ad Hoc Networks. In *Proceedings of 2002 MILCOM Conference*, Anaheim, CA, October 2002.

[IEE01] IEEE8021X. Port-based Network Access Control. IEEE Std 802.1x, 2001 Edition. IEEE Standard, June 2001.

[KV03] Pradeep Kyasanur and Nitin Vaidya. Detection and Handling of MAC Layer Misbehavior in Wireless Networks. In *to appearin Proceedings the International Conference on Dependable Sysmtes and Networks*, San Francisco, CA, June 2003.

[LB02] Mike Lynn and Robert Baird. Advanced 802.11 Attack. Black Hat Briefings, July 2002.

[Lou01] Michael Lowry Lough. *A Taxonomy of Computer Attacks with Applications to Wireless*. PhD thesis, Virginia PolyTechnic Institute, April 2001.

[Moo02] Tim Moore. Validating Disassociate Deauth Messages. Presentation to 802.11 WGi, September 2002.

[NS] NS. VINT Network Simulator. `http://www.isi.edu/nsnam/ns/`.

[Sch02] Mike Schiffman. The Need for an 802.11 Wireless Toolkit. Black Hat Briefings, July 2002.

[SIR02] Adam Stubblefield, John Ioannidis, and Aviel Rubin. Using the Fluhrer, Mantin, and Shamir Attack to Break WEP. In *Proceedings of the 2002 Network and Distributed Systems Symposium*, San Diego, CA, February 2002.

[Son] Dug Song. Dsniff Homepage. `http://naughty.monkey.org/~dugsong/dsniff/`.

Denial of Service via Algorithmic Complexity Attacks

Scott A. Crosby
scrosby@cs.rice.edu

Dan S. Wallach
dwallach@cs.rice.edu

Department of Computer Science, Rice University

Abstract

We present a new class of low-bandwidth denial of service attacks that exploit algorithmic deficiencies in many common applications' data structures. Frequently used data structures have "average-case" expected running time that's far more efficient than the worst case. For example, both binary trees and hash tables can degenerate to linked lists with carefully chosen input. We show how an attacker can effectively compute such input, and we demonstrate attacks against the hash table implementations in two versions of Perl, the Squid web proxy, and the Bro intrusion detection system. Using bandwidth less than a typical dialup modem, we can bring a dedicated Bro server to its knees; after six minutes of carefully chosen packets, our Bro server was dropping as much as 71% of its traffic and consuming all of its CPU. We show how modern universal hashing techniques can yield performance comparable to commonplace hash functions while being provably secure against these attacks.

1 Introduction

When analyzing the running time of algorithms, a common technique is to differentiate best-case, common-case, and worst-cast performance. For example, an unbalanced binary tree will be expected to consume $O(n \log n)$ time to insert n elements, but if the elements happen to be sorted beforehand, then the tree would degenerate to a linked list, and it would take $O(n^2)$ time to insert all n elements. Similarly, a hash table would be expected to consume $O(n)$ time to insert n elements. However, if each element hashes to the same bucket, the hash table will also degenerate to a linked list, and it will take $O(n^2)$ time to insert n elements.

While balanced tree algorithms, such as red-black trees [11], AVL trees [1], and treaps [17] can avoid predictable input which causes worst-case behavior, and universal hash functions [5] can be used to make hash functions that are not predictable by an attacker, many common applications use simpler algorithms. If an attacker can control and predict the inputs being used by these algorithms, then the attacker may be able to induce the worst-case execution time, effectively causing a denial-of-service (DoS) attack.

Such algorithmic DoS attacks have much in common with other low-bandwidth DoS attacks, such as stack smashing [2] or the ping-of-death [1], wherein a relatively short message causes an Internet server to crash or misbehave. While a variety of techniques *can* be used to address these DoS attacks, common industrial practice still allows bugs like these to appear in commercial products. However, unlike stack smashing, attacks that target poorly chosen algorithms can function even against code written in safe languages. One early example was discovered by Garfinkel [10], who described nested HTML tables that induced the browser to perform super-linear work to derive the table's on-screen layout. More recently, Stubblefield and Dean [8] described attacks against SSL servers, where a malicious web client can coerce a web server into performing expensive RSA decryption operations. They

[1] `http://www.insecure.org/sploits/ping-o-death.html` has a nice summary.

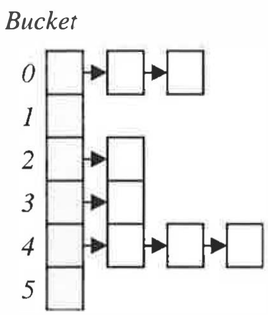


Figure 1: Normal operation of a hash table.

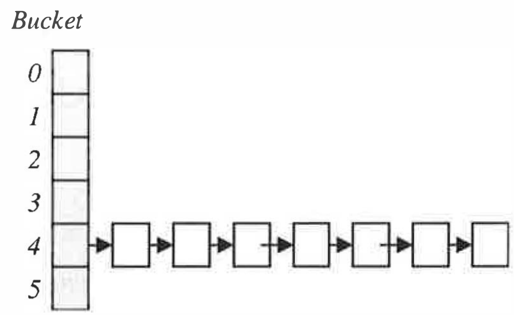


Figure 2: Worst-case hash table collisions.

suggested the use of *crypto puzzles* [9] to force clients to perform more work before the server does its work. Provably requiring the client to consume CPU time may make sense for fundamentally expensive operations like RSA decryption, but it seems out of place when the expensive operation (e.g., HTML table layout) is only expensive because a poor algorithm was used in the system. Another recent paper [16] is a toolkit that allows programmers to inject sensors and actuators into a program. When a resource abuse is detected an appropriate action is taken.

This paper focuses on DoS attacks that may be mounted from across a network, targeting servers with the data that they might observe and store in a hash table as part of their normal operation. Section 2 details how hash tables work and how they can be vulnerable to malicious attacks. Section 3 describes vulnerabilities in the Squid web cache, the DJB DNS server, and Perl's built-in hash tables. Section 4 describes vulnerabilities in the Bro intrusion detection system. Section 5 presents some possible solutions to our attack. Finally, Section 6 gives our conclusions and discusses future work.

2 Attacking hash tables

Hash tables are widely used throughout computer systems. They are used internally in compilers to track symbol tables. They are used internally in operating systems for everything from IP fragment reassembly to filesystem directory lookup. Hash tables are so common that programming languages like Perl provide syntactic sugar to represent hash tables as "associative arrays," making them easy for programmers to use. Programmers clearly prefer hash tables for their constant-time expected behavior, despite their worst-case $O(n)$ per-operation running time. After all, what are the odds that a hash table will degenerate to its worst case behavior?

In typical usage, objects to be inserted into a hashtable are first reduced to a 32-bit *hash value*. Strings might be hashed using a checksum operator like CRC32 or MD5, but are usually hashed by much simpler algorithms. More complex objects might have custom-written hash-value operators. The hash table then takes this hash value, modulo the *bucket count*, the size of the array of pointers to data being stored in the hash table, determining the bucket that will hold the reference to the object being inserted. When two inputs map the the same bucket, a *collision* has occurred. To deal with this case, each *hash bucket* holds a linked list of all inserted objects whose hash value, modulo the bucket count, maps to that particular bucket (see Figure 1). These linked lists are referred to as *hash chains*. When the total number of objects in the hash table grows too large, resulting in long average chain length, the size of the array of hash buckets is typically increased, perhaps multiplied by a constant factor, and the entries in the table are reinserted, taking their hash values modulo the new bucket count.

There are other methods of implementing hash tables, including *open addressing*, where collisions are not resolved using hash chains. Instead, the

system follows a deterministic strategy to probe for an empty hash bucket, where the object is then inserted. Although this paper focuses on hash chaining, the attacks described here will be at least as effective on open addressing hash tables.

The worse case (see Figure 2) can occur for two reasons: either the 32-bit hash values are identical, or the hash values modulo the bucket count becomes identical. Of course, for randomly chosen input, the odds of every object hashing to the same bucket is vanishingly small — $\left(\frac{1}{b}\right)^{n-1}$ for b buckets and n objects. For maliciously chosen input, however, it becomes entirely feasible. If the hash table is checking for duplicates when a new object is inserted, perhaps to guarantee that it acts as a mapping from object keys to values, then it will need to scan every entry in the hash bucket. This will induce the worst-case $O(n)$ behavior for each insert.

There are only a few requirements in order to engage in such an attack. First, the hash function being used must be deterministic and known to the attacker. Second, the attacker needs the ability to predict or supply all of the input being used by the hash function. Third, the attacker needs to ensure that a sufficient volume of attack input gets to the victim such that they experience a performance degradation.

The attacker must understand how raw data, initially read by the application from the network, is processed before it is inserted into the hash table. Knowing this, the attacker must compute objects that will eventually collide, either in the 32-bit hash-value space, or only in the eventual hash buckets. Section 2.1 will describe how these collisions can be efficiently computed for some hash functions. At worst, computing hash collisions requires an attacker to exhaustively search within the space of possible inputs. While expensive, the attacker can do this work ahead of time. Ultimately, the question is whether the victim will accept enough attack-input for the $O(n^2)$ worst-case behavior to manifest itself. Furthermore, some victims may enforce various limits on the growth of their hash tables, making them robust against this class of attack. We describe such limits in Section 2.2.

2.1 Constructing a specific attack

The first step in analyzing a program's vulnerabilities to this attack is to determine where hash tables are being used and identifying whether external, untrusted input can be fed directly into the table. This can be time consuming. As an example, the Bind DNS server places some four different abstraction layers between the network and the ultimate hash table storing DNS bindings. Tracing this can be tedious work for an attacker unfamiliar with the source code.

2.1.1 Hash collision versus bucket collision

An attacker may not know the bucket count exactly; many implementations change the bucket count based on the number of objects stored in the hash table. However, given the application's source code, an attacker may be able to guess possible values for the bucket count. This leads to two avenues of attack: those where you don't care about the bucket count and those where you know or guess the bucket count.

If collisions can be computed in the full 32-bit hash-value space, then the bucket count is irrelevant; the hash table will exhibit worst-case behavior regardless of how many buckets it has. More formally, we wish to derive inputs $k_1, k_2, \ldots k_i$ such that $Hash(k_1) = Hash(k_2) = \ldots = Hash(k_i)$. We refer to these as *hash collisions*. If the inputs have different hash values, but still collide into the same bucket (e.g., after a modulo operation has taken place), we refer to these as *bucket collisions*. Formally, a bucket collision is when we derive inputs $k_1, k_2, \ldots k_i$ such that $f(k_1) = f(k_2) = \ldots = f(k_i)$ where f is the the function mapping from inputs to buckets. In many cases, $f(k) = Hash(k) \pmod{n}$, with n being the number of buckets.

While hash collisions would seem straightforward, they do not always result in a feasible attack. For

example, consider an attacker who wishes to attack an intrusion detection system (IDS) scanning TCP/IP SYN packets to detect SYN flooding activity. If the IDS is remembering packets based purely on the source and destination IP addresses and port numbers, this would give the attacker a 96-bit search space. However, the destination address must be close enough to the IDS for the IDS to observe the traffic. Likewise, the attacker's service provider may do egress filtering that prevents forged source IP addresses. This could reduce the attacker to as little as 48-bits of freedom in selecting packets. If the hash function reduces these packets to 32-bit hash values, then there will be, on average, 2^{16} packets that an attacker can possibly send which will collide in the 32-bit hash-value space. 2^{16} values stored in the same hash bucket may or may not be enough to noticeably degrade the IDS's performance.

Conversely, suppose the attacker wishes to compute bucket collisions rather than hash collisions. Because the bucket count is much smaller than the size of the hash-value space, it will be easier to find bucket collisions. Thus, if the attacker can predict the precise bucket count, then many more possible collisions can be computed. This flexiblity may allow effective attacks on applications hashing inputs as short as 32-bits. However, if there are several possible bucket counts, then the attacker has several options:

- Guess the bucket count.
- Compute collisions that work for several different bucket counts.
- Send several streams of attack data, where each stream is designed to collide for one particular bucket count.

Computing collisions that work for multiple bucket counts is not practical; the search space grows proportionally to the least common multiple of the candidate bucket counts. This can easily exceed the 32-bit space of hash values, making hash collisions more attractive to compute than bucket collisions. However, if the number of candidate bucket counts (c) is small enough, then the attacker can compute separate attack streams focused on each potential bucket count. If the attacker sends n objects of attack data, then most of the attack data $\left(n\left(1-\frac{1}{c}\right)\right)$ will be distributed throughout the hash table, with an expected $O(1)$ insert per object. The remaining $\frac{n}{c}$ objects, however, will cause an expected $O\left(\left(\frac{n}{c}\right)^2\right)$ total running time. Furthermore, if the hash table happens to be resized and one of the attack streams corresponds to the new bucket count, then the resulting hash table will still exhibit quadratic performance problems.

For simplicity, the remainder of this paper focuses on computing hash collisions. Later, when we describe attacks against an actual IDS (see Section 4), we will show that 2^{16} collisions in one bucket are more than sufficient to mount an effective attack.

2.1.2 Efficiently deriving hash collisions

The hash functions used by typical programs for their hash tables are generally not cryptographically strong functions like MD5 or SHA-1. Instead, they tend to be functions with 32 bits of internal state, designed primarily for speed. Because this state is limited, we need only find inputs such that the internal state after hashing is the same as the initial state.

Consider a hash function with the initial state of 0. Imagine we can find *generators*, or inputs $k_1, k_2, \ldots k_i$ such that $0 = Hash(k_1) = Hash(k_2) = \ldots = Hash(k_i)$. Then the concatenation of any number of these generators in any combination and any order also hashes to 0. So, $k_1 k_2$ also hashes to 0, as will $k_1 k_1$ or $k_2 k_1 k_3 k_2$. Thus, by finding three inputs k_1, k_2, k_3 via exhaustive search and concatenating them combinatorially, we can generate a large number of collisions without requiring any additional searching. The number of possible collisions is bounded only by the maximum length to which we are willing to allow concatenated generators to grow. This process can be generalized by finding

a set of generators closed over a small number of hash states (i.e., searching for generators that take hash states less than a small integer to other hash states less than the same small integer).

In simple tests, attacking the Perl 5.6.1 hash functions on a 450MHz Pentium-2 processor, 30 minutes of CPU time enumerating and hashing all 8 character alphabetic strings was sufficient to find 46 generators that hash to zero. By concatenating three of them combinatorially, we derive 46^3 (97k) alphabetic inputs, 24 characters long, that will all hash to the same 32-bit hash value.

Hash collisions can be efficiently computed for a number of other common applications. The Linux protocol stack and the Bro intrusion detection system simply XOR their input together, 32 bits at a time. Thus, collisions may be directly computed from the algebraic structure of the hash function.

2.2 Application limits on hash tables

Many applications are sensitive about their overall memory usage, and thus have limits designed to control how large their hash tables might grow. If a hash table can never have enough elements in it for the worst-case $O(n^2)$ behavior to dominate, then our attack will fail.

2.2.1 Explicit limits

Some applications have explicit limits on their hash tables. We first consider the Linux IP fragment reassembly code. In response to earlier attacks, Linux currently allows at most 256 kbytes of storage toward reassembling incomplete packets. If we wish to attack the hash table being used to store these packet fragments, the longest hash chain we can induce will still be under 256 kbytes in total. We can still force Linux to repeatedly scan this chain, increasing the CPU load on the kernel, but we are unsure whether we can cause enough slowdown to be interesting.

(Florian Weimer reports that he found an exploitable hashing vulnerability in the Linux route cache, allowing 400 packets per second from an attacker to overload a quad-processor Pentium Xeon server, despite the size limits present in the route cache's hash table [20].)

The Apache web server collects fields from HTTP request headers into a vector (auto-sizing array). If there are multiple header fields with the same type, Apache concatenates them with an $O(n^2)$ operation. This was a target for attack [19], however Apache now imposes a limit on the number of fields that can appear in an HTTP request (100 fields, by default). Even with 100 entries naming the same field, a $O(n^2)$ worst-case running time will still be small, because n is too small for the quadratic performance to become noticeable.

2.2.2 Implicit limits

There are many other places where there are limits on the attacker's ability to influence a hash table. For instance, as discussed in Section 2.1.1, the freedom of an attacker to construct arbitrary inputs may be limited. In the case of network packets intended to attack a network sniffer, the attacker is limited both by the packet fields being watched by the sniffer, and by the packet headers necessary to route the packet toward the targeted machine. More generally, many applications operate on restricted data types, or otherwise place limits on an attacker's ability to generate arbitrary input for the targeted hash table. In some sense, these applications are lucky, but they could be vulnerable to attack in the future if their environment changes (e.g., moving from IPv4 to IPv6 will increase the size of IP addresses, giving more freedom to attack tables that hash IP addresses).

3 Application analysis: Squid, DJBDNS, and Perl

We did a short analysis of three programs to analyze how vulnerable they are to attack. We analyzed and attacked the hash tables used by two versions of the the Perl interpreter. We also analyzed and attacked the Squid web proxy cache. We investigated the DJB DNS cache and found it less vulnerable to these attacks.

3.1 Squid

The Squid Internet object cache [14] is intended to reduce network bandwidth by caching frequently used objects [7]. We analyzed the hash tables used within version 2.5STABLE1.

While we have not performed an exhaustive audit of Squid, we did discover a hash table used to track objects cached in memory. The hash table is keyed with an integer counter, the HTTP request method (i.e., GET, HEAD, etc.), and the URL in question. When Squid is operating as part of a caching cluster, it omits the integer counter and only hashes the HTTP request method and URL. (For reasons that escape us, Squid calls this "private key" vs. "public key" mode; this seems to have nothing to do with the traditional cryptographic senses of those terms.) An MD5 cryptographic checksum is performed over these values, and the resulting 128-bit value is truncated to 13 bits, identifying the hash bucket.

As an attacker, we cannot necessarily predict the value of the counter, making it difficult to compute hash collisions. However, Squid can be tricked into believing that it is part of a cluster by sending it a single UDP packet, an Internet Caching Protocol (ICP) MISS_NO_FETCH message [21]. This packet is accepted by the default configuration, and it's unclear whether this packet could be easily filtered, even using Squid's access control features. Regardless, any Squid cluster would already be forgoing the use of the "private key" mode, and thus would be vulnerable to attack.

A full benchmarking environment for Squid would require multiple web servers and clients to simulate the load experienced by the Squid web cache. To simplify things, we ran Squid on a stand-alone machine, where the URL requests were parsed from a local file and were satisfied with constant-sized web page results, served by a local proxy server. This environment is undeniably not suitable for making general remarks about Squid's general-purpose throughput, but it allows us to place pressure on this particular hash table and observe the effects.

We measured the wall-clock time necessary for Squid, in our restrictive configuration, to load approximately 143k URLs. We compared the performance of loading randomly chosen URLs with URLs carefully chosen to collide with Squid's hash function. Squid took 14.57 minutes to process the attack URLs versus 10.55 minutes to process the randomly chosen URLs. Thus, our attack added, on average, approximately 1.7ms of latency to each request serviced by the Squid cache.

This attack does not represent a "smoking gun" for algorithmic complexity attacks, but it does illustrate how common network services may be sensitive to these attacks. Furthermore, this attack demonstrates how seemingly innocuous features (e.g., Squid's "private key" mechanism, whatever it actually does) may have an effect on an application's resistance to these attacks.

3.2 DJBDNS

Dan Bernstein's DNS server is designed to have several independent programs serving different duties. His DNS cache is one program in this collection. If we can pollute the cache with requests for domains under our control (e.g., "x1.attacker.org", "x2.attacker.org", etc.), we may be able to mount an algorithmic complexity attack against the DNS cache's hash table.

Upon code inspection, DJBDNS uses a determin-

istic hash function in its implementation of a DNS cache. Interestingly, the lookup code has an explicit check for being subject to "hash flooding;" after following a chain for 100 entries, it gives up and treats the request as a cache miss. We presume this design is intended to prevent the DNS cache from burning an excessive amount of CPU on any given request. Bernstein essentially anticipated a version of our attack, although, as we discuss in Section 5, his fix could be improved.

3.3 Perl

Perl is a widely-used programming language with built-in support for hash tables (called "associative arrays"). While attacking a large number of Perl scripts is behind the scope of this paper, we expect that many deployed Perl scripts take untrusted input and store it directly in associative arrays. We demonstrate attacks against the associative arrays in Perl, versions 5.6.1 and 5.8.0; the hash function was changed between these two versions.

The hash functions in both versions of Perl form state machines. The internal state is the 32 bit accumulated hash value. The input being hashed is mixed in, one byte at a time, using a combination of addition, multiplication, and shift operations. The structure of the hash functions in both Perl 5.6.1 and 5.8.0 allow us to efficiently compute generators (see Section 2.1.2). Spending around one CPU hour attacking both hash functions, we were able to find 46 generators for Perl 5.6.1 and 48 generators for Perl 5.8.0, yielding 97k-110k colliding inputs of 24 characters in length. We then loaded these strings directly into associative arrays in both interpreters. The results are presented in Table 1. When an interpreter is fed the input designed to collide with its hash function, the running time was three orders of magnitude worse (2 seconds vs. almost two hours) than when fed the data designed to attack the other Perl version. This represents how devastating an algorithmic complexity attack can be. One hour of pre-computed CPU work, on the client, can cause almost two hours of online work for a server. Doubling the number of inputs by either finding new generators or using longer inputs would quadruple the victim's work. The exponent in the victim's $O(n^2)$ worst-case behavior is clearly dominant.

File version	Perl 5.6.1 program	Perl 5.8.0 program
Perl 5.6.1	6506 seconds	<2 seconds
Perl 5.8.0	<2 seconds	6838 seconds

Table 1: CPU time inserting 90k short attack strings into two versions of Perl.

4 Application analysis: Bro

Bro [15] is a general-purpose network intrusion detection system (IDS) that can be configured to scan for a wide variety of possible attacks. Bro is open-source and is used in production at a number of commercial and academic sites. This makes it an attractive target, particularly because we can directly study its source code. Also, given that Bro's job is to scan and record network packets, correlating events in real time to detect attacks, we imagine it has numerous large hash tables with which it tracks these events. If we could peg Bro's CPU usage, we potentially knock the IDS off the air, clearing the way for other attacks to go undetected.

In order to keep up with traffic, Bro uses packet filters [13] to select and capture desired packets, as a function of its configuration. Following this, Bro implements an event-based architecture. New packets raise events to be processed. Synthetic events can also be timed to occur in the future, for example, to track the various time-outs that occur in the TCP/IP protocol. A number of Bro modules exist to process specific protocols, such as FTP, DNS, SMTP, Finger, HTTP, and NTP.

4.1 Analysis

Bro contains approximately 67,000 lines of C++ code that implement low-level mechanisms to ob-

serve network traffic and generate events. Bro also provides a wide selection of scripts, comprising approximately 9000 lines of code in its own interpreted language that use the low-level mechanisms to observe network behavior and react appropriately. While we have not exhaustively studied the source code to Bro, we did observe that Bro uses a simple hash table whose hash function simply XORs together its inputs. This makes collisions exceptionally straightforward to derive. The remaining issue for an attack any is to determine how and when incoming network packets are manipulated before hash table entries are generated.

We decided to focus our efforts on Bro's port scanning detector, primarily due to its simplicity. For each source IP address, Bro needs to track how many distinct destination ports have been contacted. It uses a hash table to track, for each tuple of $\langle$source IP address, destination port$\rangle$, whether any internal machine has been probed on a given port from that particular source address. To attack this hash table, we observe that the attacker has 48-bits of freedom: a 32-bit source IP address and a 16-bit destination port number. (We're now assuming the attacker has the freedom to forge arbitrary source IP addresses.) If our goal is to compute 32-bit hash collisions (i.e., before the modulo operation to determine the hash bucket), then for any good hash function, we would expect there to be approximately 2^{16} possible collisions we might be able to find for any given 32-bit hash value. In a hypothetical IPv6 implementation of Bro, there would be significantly more possible collisions, given the larger space of source IP addresses.

Deriving these collisions with Bro's XOR-based hash function requires understanding the precise way that Bro implements its hash function. In this case, the hash function is the source IP address, in network byte order, XORed with the destination port number, in host order. This means that on a little-endian computer, such as an x86 architecture CPU, the high-order 16 bits of the hash value are taken straight from the last two octets of the IP address, while the low-order 16 bits of the hash value result from the first two octets of the IP address and the port number. Hash collisions can be derived by flipping bits in the first two octets of the IP address in concert with the matching bits of the port number. This allows us, for every 32-bit target hash value, to derive precisely 2^{16} input packets that will hash to the same value.

We could also have attempted to derive bucket collisions directly, which would allow us to derive more than 2^{16} collisions in a single hash bucket. While we could guess the bucket count, or even generate parallel streams designed to collide in a number of different bucket counts as discussed in Section 2.1.1, this would require sending a significant amount of additional traffic to the Bro server. If the 2^{16} hash collisions are sufficient to cause an noticable quadratic explosion inside Bro, then this would be the preferable attack.

4.2 Attack implementation

We have designed attack traffic that can make Bro saturate the CPU and begin to drop traffic within 30 seconds during a 160kb/s, 500 packets/second flood, and within 7 minutes with a 16kb/s flood.

Our experiments were run over an idle Ethernet, with a laptop computer transmitting the packets to a Bro server, version 0.8a20, running on a dual-CPU Pentium-2 machine, running at 450MHz, with 768MB of RAM, and running the Linux 2.4.18 kernel. Bro only uses a single thread, allowing other processes to use the second CPU. For our experiments, we configured Bro exclusively to track port scanning activity. In a production Bro server, where it might be tracking many different forms of network misbehavior, the memory and CPU consumption would be strictly higher than we observed in our experiments.

4.3 Attack results

We first present the performance of Bro, operating in an off-line mode, consuming packets only as fast as it can process them. We then present the latency and drop-rate of Bro, operating online, digesting

	Attack	Random
Total CPU time	44.50 min	.86 min
Hash table time	43.78 min	.02 min

Table 2: Total CPU time and CPU time spent in hash table code during an offline processing run of 64k attack and 64k random SYN packets.

packets at a variety of different bandwidths.

4.3.1 Offline CPU consumption

Normally, on this hardware, Bro can digest about 1200 SYN packets per second. We note that this is only 400kb/s, so Bro would already be vulnerable to a simple flood of arbitrary SYN packets. We also note that Bro appears to use about 500 bytes of memory per packet when subject to random SYN packets. At a rate of 400kb/s, our Bro system, even if it had 4GB of RAM, would run out of memory within two hours.

We have measured the offline running time for Bro to consume 64k randomly chosen SYN packets. We then measured the time for Bro to consume the same 64k randomly chosen packets, to warm up the hash table, followed by 64k attack packets. This minimizes rehashing activity during the attack packets and more closely simulates the load that Bro might observe had it been running for a long time and experienced a sudden burst of attack packets. The CPU times given in Table 2 present the results of benchmarking Bro under this attack. The results show that the attack packets introduce two orders of magnitude of overhead to Bro, overall, and three orders of magnitude of overhead specifically in Bro's hash table code. Under this attack, Bro can only process 24 packets per second instead of its normal rate of 1200 packets per second.

In the event that Bro was used to process an extended amount of data, perhaps captured for later offline analysis, then an hour of very low bandwidth attack traffic (16kb/s, 144k packets, 5.8Mbytes of traffic) would take Bro 1.6 hours to analyze instead of 3 minutes. An hour of T1-level traffic (1.5Mb/s) would take a week instead of 5 hours, assuming that Bro didn't first run out of memory.

Packet rate	Packets sent	Drop rate
16kb/s	192k	31%
16kb/s (clever)	128k	71%
64kb/s	320k	75%
160kb/s	320k	78%

Table 3: Overall drop rates for the different attack scenarios.

4.3.2 Online latency and drop-rate

As described above, our attack packets cannot be processed by Bro in real-time, even with very modest transmission rates. For offline analysis, this simply means that Bro will take a while to execute. For online analysis, it means that Bro will fall behind. The kernel's packet queues will fill because Bro isn't reading the data, and eventually the kernel will start dropping the packets. To measure this, we constructed several different attack scenarios. In all cases, we warmed up Bro's hash table with approximately 130k random SYN packets. We then transmitted the attack packets at any one of three different bandwidths (16kb/s, 64kb/s, and 160kb/s). We constructed attacks that transmitted all 2^{16} attack packets sequentially, multiple times. We also constructed a "clever" attack scenario, where we first sent $3/4$ of our attack packets and then repeated the remaining $1/4$ of the packets. The clever attack forces more of the chain to be scanned before the hash table discovered the new value is already present in the hash chain.

Table 3 shows the approximate drop rates for four attack scenarios. We observe that an attacker with even a fraction of a modem's bandwidth, transmitting for less than an hour, can cause Bro to drop, on average, 71% of its incoming traffic. This would make an excellent precursor to another network attack that the perpetrator did not wish to be detected.

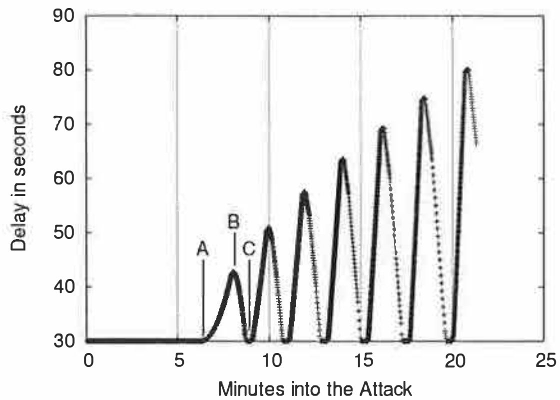


Figure 3: Packet processing latency, 16kb/s.

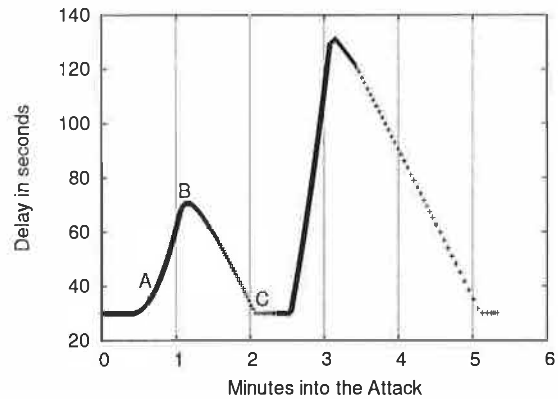


Figure 5: Packet processing latency, 64kb/s.

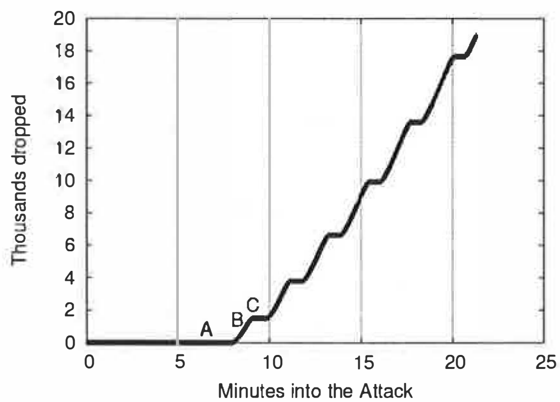


Figure 4: Cumulative dropped packets, 16kb/s.

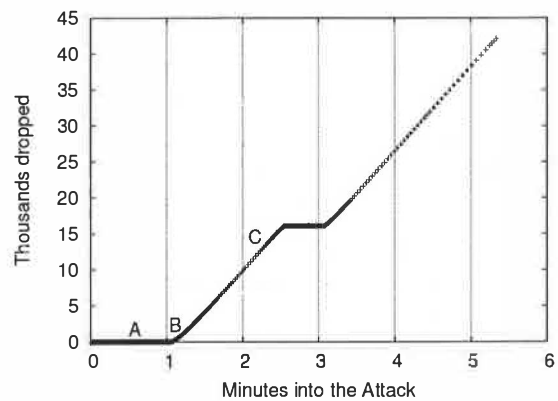


Figure 6: Cumulative dropped packets, 64kb/s.

Bro's drop rate is not constant. In fact, Bro manifests interesting oscillations in its drop rate, which are visible in Figures 3 through 6. These graphs present Bro's packet processing latency and cumulative packet drop rate for attack packets being transmitted at 16 kb/sec and 64 kb/sec.

At time *A*, the latency (time between packet arrival and packet processing) starts increasing as total processing cost per packet begins to exceed the packet inter-arrival time.

At time *B*, Bro is sufficiently back-logged that the kernel has begun to drop packets. As a result, Bro starts catching up on its backlogged packets. During this phase, the Bro server is dropping virtually all of its incoming traffic.

At time *C*, Bro has caught up on its backlog, and the kernel is no longer dropping packets. The cycle can now start again. However, the hash chain under attack is now larger then it was at time *A*. This will cause subsequent latencies to rise even higher than they were at time *B*.

This cyclic behavior occurs because Bro only adds entries to this hash table after it has determined there will be no response to the SYN packet. Bro normally uses a five minute timeout. We reduced this to 30 seconds to reduce our testing time and make it easier to illustrate our attacks. We anticipate that, if we were to run with the default 5-minute timeout, the latency swings would have a longer period and a greater amplitude, do to the ten times larger queues of unprocessed events which would be accumulated.

4.4 Discussion

Our attack on Bro has focused on its port scanning detector. Bro and other IDS systems almost certainly have other hash tables which may grow large enough to be vulnerable to algorithmic complexity attacks. For example, Bro has a module to detect network scans, determining how many destination hosts have been sent packets by a given source host. This module gives $32+h$ bits of freedom, where h is the number of host bits in the destination network monitored by the IDS. h is unlikely to be greater than 16 except for a handful of sites. However, in an IPv6 network, the sky is the limit. For that matter, IPv6 gives the attacker a huge amount of freedom for *any* attack where IP addresses are part of the values being hashed.

Part of any hash table design is the need to expand the bucket count when the table occupancy exceeds some threshold. When the hash table has a large number of objects which hash to the same bucket after the rehashing operation, then the rehashing operation could be as bad as $O(n^2)$, if the hash table were using its normal insertion operation that checks for duplicates. As it turns out, Bro does exactly this. In our regular experimental runs, we "warmed up" the hash tables to prevent any rehashing during the experiment. Before we changed our experimental setup to do this, we saw large spikes in our latency measurements that indicated rehashing was occurring. When rehashing, Bro takes 4 minutes to process a table with 32k attack entries. Bro takes 20 minutes to process a table with 64k attack entries. Without IPv6 or using bucket collisions, we cannot create more collisions than this, although making the IDS server unresponsive for 20 minutes is certainly an effective attack.

Although rehashing attacks are extremely potent, they are not necessarily easy to use; attackers cannot exploit this window of opportunity unless they know exactly when it is occurring. Furthermore, Bro's hash table will rehash itself at most 12 times as it grows from 32k entries to 64M entries.

5 Solving algorithmic complexity attacks

When analyzing algorithmic complexity attacks, we must assume the attacker has access to the source code of the application, so security through obscurity is not acceptable. Instead, either the application must use algorithms that do not have predictable worst-case inputs, or the application must be able to detect when it is experiencing worst-case behavior and take corrective action.

5.1 Eliminating worst-case performance

A complete survey of algorithms used in common systems is beyond the scope of this paper. We focus our attention on binary trees and on hash tables.

While binary trees are trivial for an attacker to generate worst-case input, many many other data structures like red-black trees [11] and splay trees [18] have runtime bounds that are *guaranteed*, regardless of their input. A weaker but sufficient condition is to use an algorithm that does not have *predictable* worst-case inputs. For example, treaps [17] are trees where all nodes are assigned a randomly chosen number upon creation. The tree nodes are rotated such that a *tree property* is preserved on the input data, as would be expected of any tree, but a *heap property* is also maintained on the random numbers, yielding a tree that is probabilistically balanced. So long as the program is designed to prevent the attacker from predicting the random numbers (i.e., the pseudo-random number generator is "secure" and is properly initialized), the attacker cannot determine what inputs would cause the treap to exhibit worst-case behavior.

When attacking hash tables, an attacker's goal is to efficiently compute *second pre-images* to the hash function, i.e., if x hashes to $h(x)$ and $y \neq x$, it should be infeasible for the attacker to derive y such that $h(y) = h(x)$. Cryptographically strong hash functions like MD5 and SHA-1 are resistant, in general, to such attacks. However, when used in hash tables, the 128 or 160 bits of output from MD5 or

SHA-1 must eventually be reduced to the bucket count, making it feasible for an attacker to mount a brute force search on the hash function to find bucket collisions. Some simple benchmarking on a 450MHz Pentium-2 allowed us to compute approximately five such collisions per second in a hash table with 512k buckets. This weakness can be addressed by using keyed versions of MD5 or SHA-1 (e.g., HMAC [12]). The key, chosen randomly when the program is initialized, will not be predictable by an attacker; as a result, the attacker will not be able to predict the hash values used when the program is actually running. When keyed, MD5 and SHA-1 become *pseudo-random functions*, which, like treaps, become unpredictable for the attacker. When unkeyed, MD5 and SHA-1 are deterministic functions and subject to bucket collisions.

5.2 Universal hashing

Replacing deterministic hash functions with pseudo-random functions gives probabilistic guarantees of security. However, a stronger solution, which can also execute more efficiently, is available. *Universal hash functions* were introduced in 1979 [5] and are cited by common algorithm textbooks (e.g., Cormen, Leiserson, and Rivest [6]) as a solution suitable for adversarial environments. It has not been standard practice to follow this advice, but it should be.

Where MD5 and SHA-1 are designed to be resistant to the computation of second pre-images, universal hash functions are families of functions (with the specific function specified by a key) with the property that, for any two arbitrary messages M and M', the odds of $h(M) = h(M')$ are less than some small value ε. This property is sufficient for our needs, because an attacker who does not know the specific hash function has guaranteed low odds of computing hash collisions.

Carter and Wegman's original construction of a universal hash function computes the sum of a fixed chosen constant with the dot product of a fixed chosen vector with the input, modulo a large prime number. The fixed chosen constant and vectors are chosen, randomly, at the beginning, typically pre-computed using a keyed pseudo-random function, and reused for every string being hashed. The only performance issue is that this vector must either be pre-computed up to the maximum expected input length, or it must be recomputed when it is used, causing a noticeable performance penalty. More recent constructions, including UMAC [4] and hash127 [3] use a fixed space despite supporting arbitrary-length arguments. UMAC, in particular, is carefully engineered to run fast on modern processors, using adds, multiplies, and SIMD multimedia instructions for increased performance.

5.2.1 Universal hash designs

Some software designers are unwilling to use universal hashing, afraid that it will introduce unacceptable performance overheads in critical regions of their code. Other software designers simply need a fast, easy-to-integrate library to solve their hashing needs. Borrowing code from UMAC and adding variants hand-optimized for small, fixed-length inputs, we have implemented a portable C library suitable for virtually any program's needs.

Our library includes two different universal hash functions: the UHASH function, submitted as part of the (currently expired) UMAC Internet draft standard [4], and the Carter-Wegman dot-product construction. We also include a hand-tuned variant of the Carter-Wegman construction, optimized to support fixed-length, short inputs, as well as an additionally tuned version that only yields a 20 bit result, rather than the usual 32 bits. This may be appropriate for smaller hash tables, such as used in Squid (see Section 3.1).

Our Carter-Wegman construction processes the value to be hashed one byte at a time. These bytes are multiplied by 32 bits from the fixed vector, yielding 40 bit intermediate values that are accumulated in a 64 bit counter. One 64-by-32 bit modulo operation is used at the end to yield the 32 bit hash

value. This construction supports inputs of length up to 2^{24} bytes. (A maximum length is declared by the programmer when a hashing context is allocated, causing the fixed vector to be initialized by AES in counter mode, keyed from `/dev/random`. Hash inputs longer than this are rejected.)

Our initial tests showed that UHASH significantly outperformed the Carter-Wegman construction for long inputs, but Carter-Wegman worked well for short inputs. Since many software applications of hash functions know, apriori, that their inputs are small and of fixed length (e.g., in-kernel network stacks that hash portions of IP headers), we wished to provide carefully tuned functions to make such hashing faster. By fixing the length of the input, we could fully unroll the internal loop and avoid any function calls. GCC inlines our hand-tuned function. Furthermore, the Carter-Wegman construction can be implemented with a smaller accumulator. Without changing the mathematics of Carter-Wegman, we can multiply 8 bit values with 20 bit entries in the fixed vector and use a 32 bit accumulator. For inputs less than 16-bytes, the accumulator will not overflow, and we get a 20 bit hash value as a result. The inputs are passed as separate formal arguments, rather than in an array. This gives the compiler ample opportunity for inlining and specializing the function.

5.2.2 Universal hash microbenchmarks

We performed our microbenchmarking on a Pentium 2, 450MHz computer. Because hash tables tend to use a large amount of data, but only read it once, working set size and its resultant impact on cache miss rates cannot be ignored. Our microbenchmark is designed to let us measure the effects of hitting or missing in the cache. We pick an array size, then fill it with random data. We then hash a random sub-range of it. Depending on the array size chosen and whether it fits in the L1 cache, L2 cache, or not, the performance can vary significantly.

In our tests, the we microbenchmarked two conventional algorithms, four universal hashing algorithms, and one cryptographic hash algorithm:

Perl	Perl 5.8.0 hash function
MD5	cryptographic hash function
UHASH	UMAC universal hash function
CW	Carter-Wegman, one byte processing, variable-length input, 64 bit accumulator, 32 bit output
CW12	Carter-Wegman, two byte processing, 12-byte fixed input, 64 bit accumulator, 32 bit output
CW12-20	Carter-Wegman, one byte processing, 12-byte fixed input, 32 bit accumulator, 20 bit output
XOR12	four byte processing, 12-byte fixed input, 32 bit output

In addition to Perl, MD5, UHASH, and three variants of the Carter-Wegman construction, we also include a specialized function that simply XORs its input together, four bytes at a time. This simulates the sort of hash function used by many performance-paranoid systems.

Figure 7 shows the effects of changing the working set size on hash performance. All the hash functions are shown hashing 12-byte inputs, chosen from an array whose sizes have been chosen to fit within the L1 cache, within the L2 cache, and to miss both caches. The largest size simulates the effect that will be seen when the data being hashed is freshly read from a network buffer or has otherwise not yet been processed. We believe this most accurately represents the caching throughput that will be observed in practice, as hash values are typically only computed once, and then written into an object's internal field, somewhere, for later comparison.

As one would expect, the simplistic XOR12 hash function radically outperforms its rivals, but the ratio shrinks as the working set size increases. With a 6MB working set, XOR12's throughput is 50 MB/sec, whereas CW12-20 is 33 MB/sec. This relatively modest difference says that universal hashing, with its strong security guarantees, can approach the performance of even the weakest hash

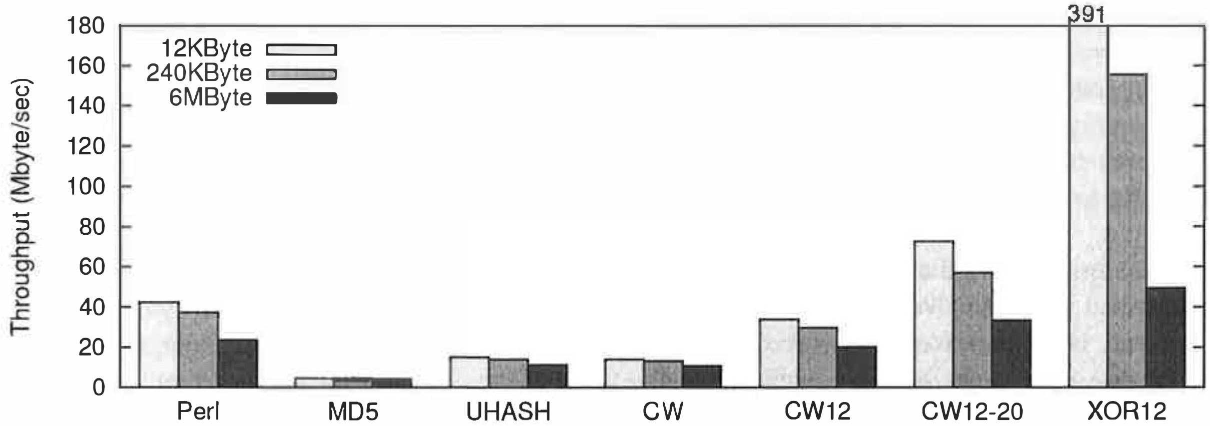


Figure 7: Effect of working set on hash performance on 12-byte inputs.

functions. We can also see that universal hashing is competitive with Perl's hash function and radically outperforms MD5.

As the cache hit rate increases with a smaller working set, XOR12 radically outperforms its competition. We argue that this case is unlikely to occur in practice, as the data being hashed is likely to incur cache misses while it's being read from the network hardware into main memory and then the CPU. Secondly, we are microbenchmarking hash function performance, a hash function is only a percentage of overall hash table runtime which is only a percentage of application runtime. Of course, individual application designers will need to try universal hashing out to see how it impacts their own systems.

Some applications hash longer strings and require general-purpose hash functions. Figure 8 uses the 6MB working set and varies the length of the input to be hashed. We can no longer use the specialized 12-byte functions, but the other functions are shown. (We did not implement a generalization of our XOR12 hash function, although such a function would be expected to beat the other hash functions in the graph.) With short strings, we see the Perl hash function outperforms its peers. However, with strings longer than around 44-bytes, UHASH dominates all the other hash functions, due in no small part to its extensive performance tuning and hand-coded assembly routines.

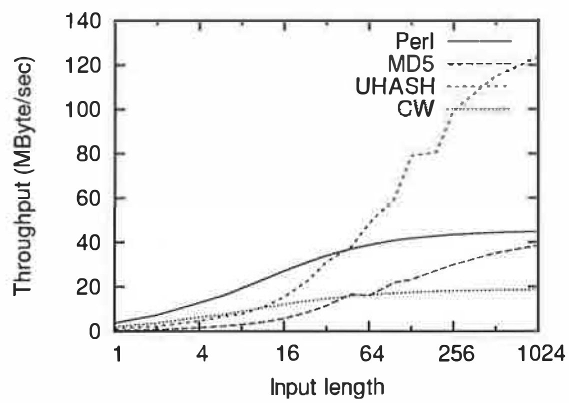


Figure 8: Effect of input length on hash performance with a 6MB working set.

We have some preliminary benchmarks with integrating universal hashing into Perl. We benchmarked the change with two perl scripts, both of which do little other than hash table operations. The first script is a histogramming program, the second just inserts text into a hash table. Our results indicate that the application performance difference between UHASH and Perl's default hash function is plus or minus 10%.

We conclude that our customized Carter-Wegman construction, for short fixed-length strings, and UHASH, for arbitrary strings, are sufficiently high performance that there is no excuse for them not to be used, or at least benchmarked, in production systems. Our code is available online with a BSD-style

license[2].

6 Conclusions and future work

We have presented algorithmic complexity attacks, a new class of low-bandwidth denial of service attacks. Where traditional DoS attacks focus on small implementation bugs, such as buffer overflows, these attacks focus on inefficiencies in the worst-case performance of algorithms used in many mainstream applications. We have analyzed a variety of common applications, finding weaknesses which vary from increasing an applications workload by a small constant factor to causing the application to completely collapse under the load cause by its algorithm's unexpectedly bad behavior.

Algorithmic complexity attacks against hash table, in particular, count on the attacker having sufficient freedom over the space of possible inputs to find a sufficient number of hash collisions to induce worst-case behavior in an application's hash table. When the targeted system is processing network packets, the limited address space of IPv4 offers some limited protection against these attacks, but future IPv6 systems will greatly expand an attacker's ability to find collisions. As such, we strongly recommend that network packet processing code be audited for these vulnerabilities.

Common applications often choose algorithms based on their common-case behavior, expecting the worst-case to never occur in practice. This paper shows that such design choices introduce vulnerabilities that can and should be patched by using more robust algorithms. We showed how universal hashing demonstrates impressive performance, suitable for use in virtually any application.

While this paper has focused on a handful of software systems, an interesting area for future research will be to study the algorithms used by embedded systems, such as routers, firewalls, and other networking devices. For example, many "stateful" firewalls likely use simple data structures to store this state. Anyone who can learn the firewall's algorithms may be able to mount DoS attacks against those systems.

[2] `http://www.cs.rice.edu/~scrosby/hash/`

Acknowledgements

The authors gratefully thank John Garvin for his help in the analysis of the Squid web proxy and Algis Rudys and David Rawlings for their careful reading and comments. We also thank David Wagner and Dan Boneh for helpful discussions on universal hashing.

This work is supported by NSF Grant CCR-9985332 and Texas ATP grant #03604-0053-2001 and by gifts from Microsoft and Schlumberger.

References

[1] G. M. Adel'son-Vel'skii and Y. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1262, 1962.

[2] Aleph1. Smashing the stack for fun and profit. Phrack #49, Nov. 1996. `http://www.phrack.org/show.php?p=49&a=14`.

[3] D. J. Bernstein. Floating-point arithmetic and message authentication. `http://cr.yp.to/papers/hash127.ps`, March 2000.

[4] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway. UMAC: Fast and secure message authentication. In *Advances in Cryptology – CRYPTO 99*, pages 215–233, 99. see also, `http://www.cs.ucdavis.edu/~rogagay/umac/`.

[5] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences (JCSS)*, 18(2):143–154, Apr. 1979.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Electrical Engineering and Computer Science Series. MIT Press, 1990.

[7] P. B. Danzig, R. S. Hall, and M. F. Schwartz. A case for caching file objects inside internetworks.

In *Proceedings of the ACM SIGCOMM '93 Conference on Communication Architectures, Protocols, and Applications*, pages 239–248, San Francisco, CA, Sept. 1993.

[8] D. Dean and A. Stubblefield. Using client puzzles to protect TLS. In *Proceedings of the 10th USENIX Security Symposium*, Washington, D.C., Aug. 2001.

[9] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. *Advances in Cryptology CRYPTO '92*, 740:139–147, August 1992.

[10] S. Garfinkel. Script for a king. HotWired Packet, Nov. 1996. `http://hotwired.lycos.com/packet/garfinkel/96/45/geek.html`, and see `http://simson.vineyard.net/table.html` for the table attack.

[11] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of 19th Foundations of Computer Science*, pages 8–21, 1978.

[12] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication. Technical Report RFC-2104, Internet Engineering Task Force, Feb. 1997. `ftp://ftp.rfc-editor.org/in-notes/rfc2104.txt`.

[13] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Annual Technical Conference*, pages 259–270, San Diego, California, Jan. 1993.

[14] National Laboratory for Applied Network Research. The Squid Internet object cache. `http://www.squid-cache.org`.

[15] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23–24):2435–2463, 1999.

[16] X. Qie, R. Pang, and L. Peterson. Defensive Programming: Using an Annotation Toolkit to Build Dos-Resistant Software. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, Boston, MA USA, December 2002.

[17] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996.

[18] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3):652–686, 1985.

[19] D.-E. C. Smorgrav. YA Apache DoS attack. Bugtraq mailing list, August 1998. `http://lists.nas.nasa.gov/archives/ext/bugtraq/1998/08/msg00060.html`.

[20] F. Weimer. Private communication, Apr. 2003.

[21] D. Wessels and K. Claffey. Application of Internet Cache Protocol (ICP), version 2. Technical Report RFC-2187, Internet Engineering Task Force, Sept. 1997. `ftp://ftp.rfc-editor.org/in-notes/rfc2187.txt`.

Plug-and-Play PKI: A PKI your Mother can Use

Peter Gutmann
University of Auckland

Abstract

A common complaint about PKI is that it is simply too hard to use at the end-user level. Somewhat surprisingly, there exists no PKI equivalent of DHCP or BOOTP for automated, transparent PKI setup, leaving the certificate user experience similar to the process of bringing up an X.25 link. This paper provides a PKI equivalent of these basic bootstrap services that provides automatic, transparent configuration and setup of certificate information, with the user needing to supply no more than their user name and password. The work covers the design process involved and tradeoffs made, implementation details, and experiences with actual operation. The overall purpose of the work is to design and implement a plug-and-play certificate setup mechanism usable by even the most inexperienced user, a "PKI your mother can use".

1 Introduction

Despite many years of effort, PKI technology has failed to take off except in a few niche areas. Reasons for this abound, and include the difficulty of deploying the technology, cost, lack of interoperability, and the poor match of PKI designs to any pressing real-world problems. Probably the primary factor at the user level though is the high level of difficulty involved in deploying and using a PKI.

There is considerable evidence from mailing lists, Usenet newsgroups and web forums, and directly from the users themselves, that acquiring a certificate is the single biggest hurdle faced by users[1]. For example various user comments indicate that it takes a skilled technical user between 30 minutes and 4 hours work to obtain a certificate from a public CA that performs little to no verification, depending on the CA and the procedure being followed. Obtaining one from non-public CAs that carry out various levels of verification before issuing the certificate can take as long as a month. A representative non-technical user who tried to obtain an (unverified) certificate from a public CA took well over an hour for the process, which involved having the author tell them where to go to start the process, filling out eight (!!) browser pages of information, having to spend time finding their passport, several restarts due to values being rejected, waiting for emailed instructions, numerous questions to the author about various points of the process, cutting and pasting data values from an email message to a web page, and filling in more information over a succession of eleven further web pages (including several that were completely incomprehensible to the user, she just clicked "Next"). Eventually, a final page announced that a certificate had been issued, although the user was never able to locate it on the PC, and couldn't understand much of the information on the web page, which referred to things like "certificate Distinguished Names" and "X.509 SubjectAltName" (there was a button labelled "Fetch", but this seemed to have no effect). Eventually an emailed message provided a URL to download the certificate, which, after switching to a different web browser and providing further names and passwords yielded a file on disk which the user located after considerable searching. Clicking on the file had no effect, at which point the user gave up.

In contrast, obtaining the Internet connection that enabled the certificate process took around ten minutes and involved the exchange of a user name, password, ISP phone number, and a credit card number (the sample certificate was free, or the process would no doubt have taken even longer). Similarly, connecting a new system to a LAN usually requires no more than a user name and password for access to the appropriate server, with the rest being taken care of transparently via mechanisms such as DHCP. In contrast, the process of obtaining a certificate most closely resembles the experience of bringing up an X.25 link. The usability problems that arise in such a system are summed up by a book that examines the reasons for the success of peer-to-peer software: "the vast majority of users detest anything they must configure and tweak. Any really mass-appeal tool must allow an essentially transparent functionality as default behaviour; anything else will necessarily have limited adoption" [1].

The problems that this creates are demonstrated by what happens when technically skilled users are required to work with certificates. The OpenSSL toolkit [2][3] includes a Perl script `CA.pl` that allows users to quickly generate so-called clown suit certificates (ones that "have all the validity of a clown suit" when used for identification purposes [4]), which is widely-used in practice. The cryptlib toolkit [5][6] contains a similar feature in the form of Xyzzy

[1] The use of anecdotal evidence in this paper is an unfortunate necessity because few public PKI usability studies exist, and the few that do tend to be post-mortem analyses.

certificates (added with some resistance and only after the author grew tired of endless requests for it), ones with dummy X.500 names, an effectively infinite lifetime, and no restrictions on usage. Most commercial toolkits include similar capabilities, usually disguised as "test certificates" for development purposes only, which end up being deployed in live environments because it's too difficult to do it the way X.509 says it should be done. Certificates used with mailers that support the STARTTLS option consist of ones that are "self-signed, signed-by the default Snake Oil CA, signed by an unknown test CA, expired, or have the wrong DN" [7]. The producer of one widely-used Windows MUA reports that in their experience 90% of the STARTTLS-enabled servers that they encounter use self-signed certificates [8]. This reduces the overall security of the system to that of unauthenticated Diffie-Hellman key exchange, circa 1976. In all of these cases, the entire purpose of certificates has been completely short-circuited by users because it's just too difficult to do the job properly. The problematic nature of X.509 is echoed in publications both technical and non-technical, with conference papers and product descriptions making a feature of the fact that their design or product works without requiring a PKI. For example, one recent review of email security gateways made a requirement for consideration in the review that the product "have no reliance on PKI" [9]. As an extreme example of this, the inaugural PKI Research Workshop, attended by expert PKI users, required that submitters authenticate themselves with plaintext passwords because of the lack of a PKI to handle the task [10][11].

The goal of this paper, then, is to design and build a certificate-handling mechanism whose use is no more difficult than the task of obtaining an ISP account. Somewhat less formally, the intent is to "build a PKI your mother can use". Since ISPs have had many years of experience in making what was originally a complex operation best left to experienced technical users accessible to everyone, this paper uses the approach used by ISPs to solve similar problems as a starting point for solving equivalent problems from the PKI field.

2 The Problem

Security texts generally recommend the following security principles for good key management [12][13][14]:

- Encryption (as opposed to signature) keys should have as short a lifetime as possible in order to shorten their window of vulnerability. Changing keys frequently reduces the damage if a key is compromised.
- Different keys should be used for different purposes. For example one (short-term) key might be used for encryption, a (different) medium-term key for ephemeral authentication and a long-term key for digital signatures that need to be validated years after signing. Even in the latter case, only the public portion of the key should be retained over the long term for signature verification, with the private portion being destroyed to reduce potential exposure.
- Keys should be carefully protected, and never distributed

A consequence of the difficulty in obtaining certificates is that users break every principle of good key management in order to avoid the pain of acquiring/replacing a certificate [15]:

- CAs use certificates with effectively infinite lifetimes (20-40 years).
- Users use certificates with effectively infinite lifetimes (even if the CA limits it to 1 year for billing purposes) by re-certifying the same key year in, year out. Many users see this as perfectly natural and sensible, and some CAs concur.
- The same key is used for everything (encryption, long-term signing, ephemeral authentication, etc, etc). This is regarded as a CA and/or software feature by many users. For example, the Windows CryptoAPI allows an encryption-only key to be used for signatures as well, since this is more convenient for users than having to manage two keys.
- The private key is copied around freely so that every application on the system and in some cases every user or machine on the network can share it. Again, the ability to freely distribute the private key is regarded as a software feature.
- In some cases, this effect can even stall the deployment of newer, more secure mechanisms to replace older ones with known security problems. One of the contributing reasons why RSA with PKCS #1 padding is still in almost universal use rather than being replaced with a more secure alternative such as RSA OAEP or RSA PSS is that it's too difficult to update the certificates involved [16].

In all of these cases, the pain of certificate acquisition and renewal has overridden all security concerns. The problem we face, then, is how to make certificate acquisition and update as transparent and simple as possible. The best way to determine what needs to be done is to examine the individual issues raised by the

certificate acquisition example given earlier, and provide representative solutions as used in the ISP world.

2.1 Initial Assumptions

Before we begin, we need to define the environment within which we're operating. The work presented here makes the following basic assumptions:

- There is an existing relationship between the user and some entity involved in the certification process that can be used for authentication purposes. For example when a user signs up for online banking or a loyalty program, the bank or vendor typically sends them an authenticator for the online enrolment in the post, or communicates it over the phone or in person. A similar out-of-band exchange of initial authentication information is assumed here. This follows existing practice and uses well-established mechanisms, which is important not only for obvious practical reasons but also because it comes with valuable legal precedent in case of a dispute.
- There is a server available that provides the certification services discussed further on, or at least the PKI service-locator service. This is the PKI equivalent of a DHCP server. Just as with a DHCP server, a user can still manually configure alternatives if they so desire or if the server is unavailable. The server may provide the services directly, for example as part of an ISP's overall service package or through an in-house CA, or it may merely act as a front-end or forwarder to another service provider such as a traditional public CA. For example it's quite possible that a large service provider such as AOL or MSN may want to run a CA as a value-added service for users. Similarly, large organisations such as corporates, universities, and government departments may run their own CAs. Smaller organisations, on the other hand, would typically farm the service out to an external CA.
- We're not designing a system to handle nuclear weapons launch codes. The system need only be as secure as an equivalent non-PKI alternative: "Cumbersome technology will be deployed and operated incorrectly and insecurely, or perhaps not at all" [17]. For example, typical online operations for which a certificate might be used (shopping, banking, vendor loyalty programs) all get by with a user name and password as the basic input to the authentication process. If it's good enough for existing applications, it's good enough here, although we use more appropriate cryptographic support (message authentication codes (MACs) and digital signatures) than the usual passwords-over-SSL.

The requirements that need to be met by the enrolment process are currently the subject of an extensive public consultation process being performed by the New Zealand State Services Commission [18], who have been soliciting input from various sources including government departments, businesses, and end users, as well as special-interest groups such as the New Zealand Privacy Commissioner. The work also takes into consideration similar work being carried out in Australia, Canada, Ireland, Singapore, and the UK, and will be the subject of a future paper.

The consultation process has among its policy goals producing an enrolment system which is acceptable to potential users (taking into account the different needs of different users), providing appropriate privacy protection, and being fit for its purpose, avoiding unnecessary over-engineering. The implementation goals include providing a user-focused system which is as convenient, easy to use and non-intrusive as possible, being flexible enough to accommodate change, being affordable and reliable for the public and for government agencies, and very importantly complying with relevant law, including privacy and human rights law, and functioning in a way that provides legal certainty. A final implementation goal is that it provide functional equivalence to existing systems, so that the authentication requirements are similar to those that apply to existing transactions except in cases where the online nature of the transaction significantly changes the level of risk. Although the State Services Commission is not considering a PKI-based enrolment system for individuals, the principles that underpin their work can be adopted to develop a plug-and-play PKI.

Now that the initial assumptions have been laid out, we can examine the various design problems that we need to address.

2.2 Locating the Certificate Service

The user was required to manually locate the certification service (typically a Certification Authority or CA) that they were using. In contrast a protocol such as DHCP automatically locates the services required for Internet access without the user even being aware that this is happening.

2.3 Entering User Data

Once the user had located the CA service, they were required to enter a considerable amount of data before they could go any further. In particular, despite the fact that the certificate being requested would contain no

more than an (unverified) name and an email address (the rest is fixed data set by the CA), the user was required to fill in pages and pages of information unrelated to the certificate. After providing authentication data, the user was emailed back multiple authenticators that had to be cut and pasted into web pages and then had to answer several more pages of questions before a certificate was issued. Imagine the loss of productivity faced by any organisation that has its employees enrol for certificates if each employee is forced to run this gauntlet!

In contrast, signing up for an ISP account or connecting to a corporate network requires a single user name (to identify the user) and a password (to authenticate them). Even processes such as online banking (with far more at stake then a zero-value email certificate) can get by with a simple name and authenticator.

2.4 Obtaining the Certificate

The process of obtaining the certificate once it was issued was problematic and difficult for the user to comprehend, both conceptually (the user was told that a certificate had been issued, but all that meant was that a certificate existed somewhere, not that the user actually had it), and practically (it didn't work properly). In contrast, the success or failure of the ISP signup process is immediately obvious, and requires no further action by the user (barring the usual networking snafus).

2.4.1 PKI-enabled Toasters

A slightly different situation occurs when the certificate user isn't a human being but a device, particularly a device with a very limited user interface that can't function as, or emulate, the behaviour of a human being: the Internet-enabled toaster. Designing mechanisms to make working with these devices possible is the goal of zero-configuration networking [19][20][21], inspired by the host-requirements RFC observation that a self-configuring networking system would permit "the whole suite to be implemented in ROM or cast into silicon [...] it would be an immense boon to harried LAN administrators as well as system vendors".

The ability to initialise a PKI-enabled toaster is an additional plug-and-play PKI service that has no equivalent in the ISP world.

2.5 Bootstrapping the PKI

A final problem, which was hidden by the use of the web browser, is obtaining the initial CA certificates that are used to verify other certificates. Web browsers address this problem by including a large (well over 100 certificates for current browsers) collection of certificates hardcoded into them. These hardcoded certificates include ones from totally unknown CAs (including ones whose private keys have been on-sold to various third parties [22]), CAs with moribund web sites, 512-bit keys, policies disclaiming all liability for any use of the certificate or reliance on information contained in it, and similar worrying signs [23]. Furthermore, because the browser trusts all CA certificates in its collection equally, the overall system is only as trustworthy as the least trustworthy CA in the collection. In other words, Honest Joe's Used Cars and Certificates is assigned the same status as the Verisign Class 1 Public Primary Certification Authority, and can usurp its certificates if it so desires.

This bootstrap process, referred to in this paper as PKIBoot, is a PKI-specific issue that has no equivalent in the ISP world.

2.6 Problem Summary

In summary, we need a system with the following features:

- Transparent discovery of PKI services
- Bootstrap functionality that doesn't assume pre-existing trusted certificates
- Simple (user name + password) enrolment process
- Automated acquisition and renewal of certificates

Section 3 examines possible mechanisms that may be used to provide these services, and sections 4 and 5 go on to cover the actual implementation, and implementation experiences.

3 Approach

In order to meet the requirements listed previously, we need three distinct mechanisms:

- A service location mechanism
- A bootstrap mechanism
- A certificate acquisition/renewal mechanism

The abstract flow of operations for this process is shown in the sequence diagram in Figure 1. Initially, the user needs to determine where to go to obtain certificate services. This is handled by the service location step. Employing security mechanisms at this very preliminary level would be rather problematic since the actual service that handles security hasn't been engaged yet. There is in fact no need for this service to be secure, since the interaction with the certificate service that follows is authenticated, in the same way that SSL (in theory) doesn't require a secure DNS service since it authenticates the client and server.

In practice though, SSL authenticates the server using a certificate from one of over a hundred CAs hardcoded into the client software that the user has no choice but to trust, and omits user authentication entirely because it's just too hard to manage, providing a good example of the exact problem that the plug-and-play PKI services are intended to solve.

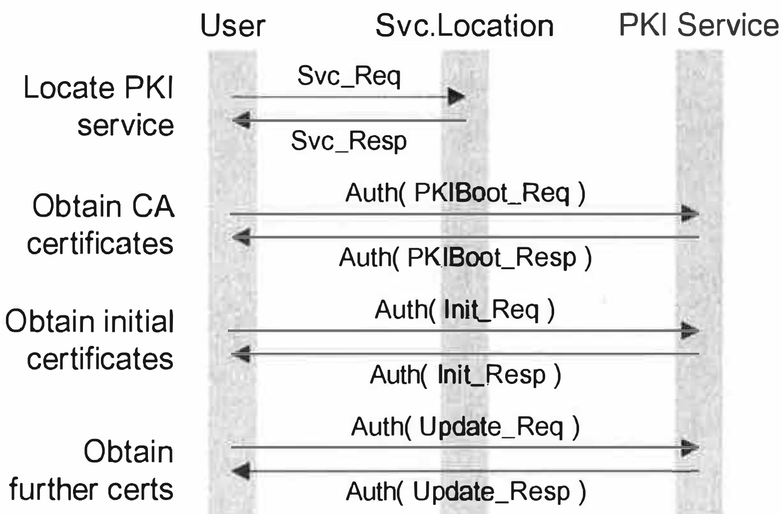


Figure 1: Plug-and-play PKI setup process

Once the user has located the certificate service provider, they cryptographically authenticate themselves to the server via the mechanism described in section 4.3 and request an initial set of certificates such as trusted CA certificates. For example if the certification service were being run by the user's employer, the server would return the employer's CA certificates (and any other certificates that the employer considers trustworthy, for example those of trading partners) to the user, authenticated using the same mechanism as the one used by the user. This is equivalent to the certificate collection normally hardcoded into web browsers, except that the user is accepting a small set of certificates from the trusted service provider rather than a large collection of arbitrary certificates from sources completely unknown to them.

Now that the user has the necessary CA certificates, they can obtain their first certificate, identified by their user name and authenticated by their password (again, used as input to the appropriate cryptographic mechanism). In the case of the Internet-enabled toaster where there's no password available, the authentication is provided by a unique ID such as the toaster's serial number (which doesn't have to be private, merely unique) and the use of a private network segment for initialisation (in other words the security controls at this initial level are physical rather than electronic, exact details of the process are given in section 3.2.1).

Finally, the user can request further certificates using either their user name and password as before, or (more conveniently and logically) using the initial certificate to authenticate further requests. In the case of the Internet-enabled toaster, further operations can now be carried out off the private network, with communications authenticated using the certificates exchanged during the PKIBoot stage. Further certificates can be now obtained, and existing ones updated at any time.

Rather than designing yet another new PKI protocol (the IETF PKIX working group currently has 21 RFCs and 30 RFC drafts in the works, with several being small books over 100 pages long, for an incredible 1,600 pages of X.509 PKI standards), our intent is to make use of existing protocols and mechanisms wherever possible (although no standard exists that provides details on the basic act of initialising a PKI via a PKIBoot-type mechanism, there do exist RFCs covering issues such as how to add animations and theme music to an X.509 certificate [24]). An additional goal, not stated explicitly above, is to aim for a solution which is practical and workable (a single paper suffices to cover it), rather than the ultimate perfect certificate management mechanism (260 pages of RFCs in 4 parts with 6 additional RFC drafts in progress and a 200-message thread arguing over what colour to paint it when it's finished).

3.1 PKI Service Location

The first requirement is for a protocol to locate PKI services. In the comparison with signing up for an ISP account or connecting to a corporate network that was used in section 2, mention was made of the use of DHCP for locating IP-related services, so the obvious solution would be DHCP. However, there are also a number of other service-location mechanisms that should be considered, of which the major ones are Jini, UPnP, the Service Location Protocol (SLP), and perhaps a few lesser-known ones such as Salutation.

3.1.1 DHCP

The Dynamic Host Configuration Protocol [25] provides a framework for machines on a network to obtain information such as their IP address, subnet mask, and default gateway. Aside from having

computer-literate relatives, DHCP is the primary mechanism used to make IP work automagically for users.

Use of DHCP for PKI service location has several problems. Many networks already contain DHCP servers, and don't take kindly to the appearance of a second server: "The diversity of hardware and protocol implementations in the Internet would preclude reliable operation if random hosts were allowed to respond to DHCP requests" [25]. Overloading existing servers for use for PKI service location would represent a nightmare for sysadmins and home users, who would be faced with reconfiguring an arbitrary variety and number of DHCP servers, many of which don't allow user configuration of the kind required for our purposes. Finally, DHCP functions at a very low level, while the PKI services can assume that an IP networking infrastructure is already up and running, doing away with the need for the low-level DHCP approach. All of these issues make DHCP unsuited for our purposes, apart from providing the bare plug-and-play conceptual model for our design.

3.1.2 Jini

Jini is used to locate and interact with Java-based services [26][27]. It has the disadvantage that it is tied to a particular programming language and philosophy, and requires a large amount of Java-specific mechanisms such as Java object serialisation, RMI (Remote Method Invocation), and code downloading (the ability to move Java objects between JVMs) in order to function. In addition it provides a large range of client/server communications services that are unnecessary for our purposes, since our intent is to use a standard PKI protocol if possible rather than implementing our own one using Java. This makes Jini unsuited for our purposes.

3.1.3 UPnP

UPnP is XML's (or perhaps Microsoft's) answer to Jini [28][29]. Like Jini, it provides a mechanism for locating and interacting with services over a network. Also like Jini, the protocol is very complex, employing XML (SOAP) over HTTP [30], and provides a large range of capabilities such as the ability to query devices down to the level of manufacturer details, serial numbers, and UPC codes, an HTML GUI interface, URLs for assorted control interfaces, and so on and so on. In addition the protocol is peer-to-peer and aimed mainly at home environments, allowing devices such as printers, scanners, and cameras to communicate with a computer and each other without requiring tedious user configuration.

There exists a service-discovery subset of UPnP, the Simple Service Discovery Protocol (SSDP), which operates on HTTP over UDP and restricts itself purely to service discovery. This protocol provides more or less the same services as SLP (see section 3.1.4) and bootstraps directly into SOAP/GENA (the UPnP General Event Notification Protocol), so it's not really useful as a standalone service location protocol, and in any case doesn't provide any real advantages over SLP. A more pragmatic consideration is that sites may block UPnP from leaving the LAN or disable it at the host for security reasons, in the same way that they currently block/disable NetBIOS. This would make services such as PKIBoot collateral damage even though it isn't really a UPnP service. All of these considerations make UPnP unsuited to our purposes.

3.1.4 SLP

The Service Location Protocol (SLP) is an IETF standards-track protocol designed to function in a language and technology-neutral manner [31][32][33][34]. SLP is purely a lookup service that provides a service location mechanism and then gets out of the way, which appears to make it an ideal choice for a PKI service location mechanism. However, there are some issues that make it slightly less than ideal. Firstly, the protocol is quite complex to implement, although this is mitigated to some extent by the existence of a freely-available reference implementation [35]. A more significant concern is that SLP is still waiting for its killer application (PKI is unlikely to be that killer application), so there is little deployment or even knowledge of its existence. In order to avoid requiring sysadmins to install, configure, and maintain yet another network service, we allow for the use of SLP where available, but also provide a simple alternative in section 3.1.6.

3.1.5 Others

There exist a variety of other, lesser-known service discovery protocols such as Salutation [36] and Apple's Rendezvous [37]. Although these have advantages for particular environments (for example Rendezvous brings AppleTalk-style ease of use to TCP/IP environments, which is of benefit to Macintosh users), they have no particular advantage over SLP for our purposes. In addition there are efforts under way to provide bridging for some of the more common service location protocols, such as having Salutation able to query SLP servers [38]. It appears that SLP (if it takes off) will be the common-denominator protocol.

Another possible alternative is the use of DNS SRV records [39]. Unfortunately the operating system used by the user group most in need of handholding when it

comes to technical issues has no support for anything beyond the most basic DNS address lookups, making it impossible to use DNS SRV with anything but very recent Win2K and XP systems. To make things even more entertaining, several of the function names and some of the function parameters changed at various times during the Win2K phase of development (SRV support was introduced for Active Directory under Win2K), and the behaviour of portions of the Windows sockets API changed in undocumented ways to match. This leads to the unfortunate situation in which a Unix sysadmin can make use of DNS SRV to avoid having to deal with technical configuration issues, but a Windows'95 user can't. As a result, we can't usefully rely on SRV for our needs.

3.1.6 Faking it

In the absence of any formal service-location mechanism, it's possible to fall back to the use of a "well-known" location constructed from the service provider's domain name or the device's IP address [40]. If the PKI service isn't provided locally, HTTP type 3xx redirection or DNS-based redirection may be used to redirect clients to the actual PKI service provider. Use of this means of service location has the significant feature that it can be managed through a single line of DNS or HTTP configuration data added to existing services/servers, making it by far the easiest option to deploy of any of the service location mechanisms — the addition of a single DNS entry for, for example, `pkiboot.aol.com`, would instantly provide plug-and-play PKI facilities for the entire AOL user base, with no further configuration necessary[2].

When used with full-scale DNS-enabled IP, the well-known location is constructed by prepending "pkiboot" to the domain name to obtain the URL to use for PKI services. For example if the current operating environment has a domain `myorg.org` then the PKI service would be accessed as `http://pkiboot.myorg.org`. When the certificate service is being provided in a more limited form such as with a web-enabled embedded device, the access location is given by appending a fixed path portion "pkiboot" to the device's IP address or location. For example if the embedded device providing the service is available at 192.0.0.1 then the PKI service on the device would be accessed as `http://192.0.0.1/pkiboot/`.

In practice we can use an extra level of indirection to locate PKI services, since PKIBoot isn't the only service we require, and the services may be distributed across multiple servers for performance, maintainability, or security reasons. For example the server that publishes certificate revocation lists (a small number of very large files, often several MB in size) has very different operational characteristics from the one that makes certificates available (a large number of very small files), which in turn has very different security characteristics than the one that provides certificate status information by directly querying the CA certificate store.

Since this is the part of the system that is the most likely to be maintained manually by an overworked sysadmin, we make it as simple as possible: A set of entries matching the required PKI service to the server URL that provides it, one service per line. This is in effect a crude form of DNS SRV functionality that can be made to work with versions of Windows earlier than XP. Note that no authentication is necessary at this preliminary pre-PKI level — the one thing that a PKI facility doesn't lack is endless capability for authentication and authorisation at every stage of the process.

A possible alternative exists in the form of the authorityInfoAccess (AIA) and subjectInfoAccess (SIA) certificate extension, which can be used to indicate the location of various certificate services. Since URLs can be fairly ephemeral while certificates tend to persist forever, experience has shown that these URLs invariably point to servers that no longer exist (at one point it was suggested that certificate standards be reworded to require that certificates contain only invalid URLs, to match existing practice [41]), not helped by the fact that the entity that issues the certificates usually has different goals from the one that runs the servers that provide various associated services. Since no CA will reissue all of its certificates simply because a server URL has changed, we can't depend on the AIA/SIA extension, although we can opportunistically use it if it's present and actually valid.

3.2 PKIBoot

There currently exists no protocol or mechanism for this fundamental PKI operation. Since one of our design goals was to use existing message formats if at all possible rather than inventing yet another new PKI protocol, we need to locate an existing mechanism that can be adapted to our needs. This task is made easier (or perhaps more complex) by the fact that every new PKI protocol seems to invent its own message format incompatible with that of all other protocols, providing us with over a dozen different ones to choose from. All of these implement simple request/response mechanisms that provide more or less the same

[2] The question of whether exposing AOL users to certificates is a sound idea is beyond the scope of this paper.

functions, so in theory any would fit the bill. In order to make things easier, we focus only on the two that are specifically designated as general-purpose certificate management protocols, with a discussion of their relative merits provided in the next section, section 3.3.

In order to support the PKIBoot operation, we use the GenMsg (General Message) facility of the Certificate Management Protocol (CMP, discussed further in section 4.3) to provide the certificate setup operation shown as the second stage of Figure 1 (CMP also includes a kitchen-sink field in the message header, but this appears to be dedicated to workarounds for problems in the initial version of the CMP protocol, so we use a GenMsg instead. In addition, the CMP RFC at one point appears to make a reference to a vaguely PKIBoot-type of operation, but never defines any messages or data types to support it, so this can't be implemented as a standard part of the protocol). A PKIBoot request consists of a CMP GenMsg containing a request for the initial set of trusted certificates, authenticated with a MAC (Message Authentication Code, a cryptographic checksum derived from the user's enrolment password) for new users or a digital signature for existing users who already have a certificate.

The CA responds to the PKIBoot request with the initial trusted certificate set, also protected with a MAC or a signature. In this way the user will accept only a restricted set of known-good certificates from a cryptographically authenticated authority, rather than an arbitrary collection of certificates included by whoever supplied their web browser or PC. This authentication process also eliminates the need for cryptographically protecting the service discovery stage that was covered in section 3.1, since only the genuine CA will be able to generate the MAC/signature necessary to authenticate the initial certificate set (the CMP protocol includes additional features such as nonces to prevent replay attacks, transaction IDs to link messages, and so on. Interested readers are referred to the CMP specification for more information).

3.2.1 PKIBoot for PKI-enabled Toasters

The design presented here works just as well for Internet-enabled toasters as it does for devices with human users through the use of the baby-duck security model [42][43]. In this model, a newly-initialised device (either one fresh out of the box or one reset to its ground state) imprints upon the first device it sees in the same way that a newly-hatched duckling imprints on the first moving object it sees as its mother. In our case the device trusts the first entity it meets merely to issue it a certificate, not unconditionally as in the original baby-duck model. The process is shown in Figure 2.

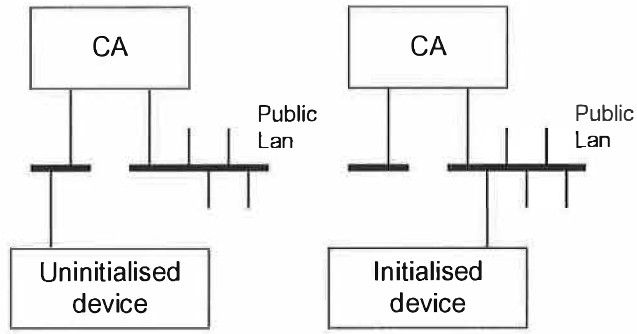


Figure 2: PnP PKI setup for an embedded device, with PKIBoot (left) and further setup (right)

Initially, the device is in the uninitialised (or reset-to-zero) state. Upon being connected to a network for the first time, it goes through the PKIBoot process to obtain its CA certificates and an initial certificate for itself, as shown in the left half of the diagram. This initialisation is carried out on a private LAN dedicated to device initialisation. Once the device has imprinted on the certificate server, it is moved to the public LAN (or WAN) and carries out any further operations there, as shown in the right half of the diagram. The initially configured state can be used to manage operations such as secure firmware uploads and generation of upload receipts by the device [44]. Note that we don't touch on the issue of who the device should trust (this is a making-PKI-workable paper, not an embedded device security paper). Readers are referred to the original work on the baby-duck security model for more coverage of this area [42][43].

Although Figure 2 shows a single server providing service for both sides, this is merely a convenience; the private network segment used for initialisation may have a server dedicated purely to device initialisation. Consider for example the case of an engine management system (EMS) of a car using the baby-duck PKI model. The initialisation stage is carried out at the manufacturer before the car is shipped to dealers. Once in the field, the EMS can use the initial certificate set to verify that upgrades are only performed by authorised service agents, and use its certificate to authenticate itself to the service agent[3].

[3] As a side-benefit of these measures, an army of hot-chip enthusiasts will provide the system with considerable penetration testing at no cost to the vendor.

3.3 Obtaining Certificates

There exist two different protocols for certificate management, Certificate Management Protocol or CMP [45] and Certificate Management Messages over CMS or CMC [46]. Both are quite complex, and their main difference is that CMP has the curious design goal of being deliberately incompatible with any standard message format (really!) [47] while CMC was designed to use existing, well-established formats [48]. Some of the consequences of the CMP design decision are examined in more detail in sections 4.3 and 5.2. At the moment, CMP seems to be winning (in the sense that CMP is rarely used while CMC appears to be unused), so we employ a carefully-profiled subset of CMP here.

4 Implementation

This section looks at the implementation details of the various protocols and mechanisms chosen in section 3. The section that follows examines specific issues that cropped up during the process, and analyses the results of the implementation.

4.1 PKI Service Location

Implementation of the service location step was simple and straightforward, both for SLP and for the DNS/HTTP-based alternative. SLP required building the OpenSLP server and client and configuring the server with a simple template for PKI services, accessible via the SLP URL `service:pki.test:-http://certificates.myorg.org`. This URL can be built automatically using the rules from section 3.1.6. The alternative option, using DNS and/or HTTP to provide an equivalent service, does more or less the same thing, but without requiring SLP as an intermediary.

This illustrates the major problem arising from the lack of SLP deployment mentioned in section 3.1.4: SLP is a type of directory service which assumes that users will contact a well-known, central directory service (a Service Agent or Directory Agent in SLP terminology) to handle requests for particular services. This works well when SLP is in general use for locating all manner of services, but when the only service in use is the PKI service, having to find the SLP service in order to find the PKI service is an unnecessary complication. Although there is a facility for locating SLP services via DHCP [49], this has the same problems as using DHCP to directly locate the PKI service that were discussed in section 3.1.1 (SLP can also use UDP multicast for discovery, but this isn't a general-purpose solution since its range rarely extends beyond the local network segment).

In addition, SLP is a powerful, general-purpose service location protocol capable of responding to very specific requests such as "Where do I get Slowaris drivers for a WalletBuster 5000 colour printer?", when all we really need is "Where's the certificates?". Because of all of these issues, in practice it's easier to perform the PKI service location via the DNS/HTTP mechanism than via SLP, although we retain the ability to use SLP if it becomes generally adopted (this is no doubt the same excuse being used by every other not-quite user of SLP).

4.2 PKIBoot

The PKIBoot implementation was quite straightforward, and required only a relatively minor modification of the CMP implementation in the cryptlib toolkit [6] to handle the new GenMsg subtype. When in use, the certificate server is initialised with a set of trusted certificates that, at a minimum, includes its own certificate(s) and/or the certificates of the CA on whose behalf it operates. This operation is handled in a fairly straightforward manner as part of cryptlib's role-based access control system.

cryptlib recognises multiple user roles that are assigned different capabilities, one of which is a CA role authorised to issue and revoke certificates. Each user role has a variety of parameters associated with it, one of which is a collection of certificates used/trusted by the user. When the server receives a PKIBoot request, it queries the CA user for its trusted certificates, and sends these as the response to the PKIBoot query. The server verifies an incoming request using the authentication information it has stored for the user making the request, and responds with the initial trusted certificate set, exactly as described in section 3.2.

Because of the automated role-based handling of certificates (the CA will, at an absolute minimum, have its own certificate(s) trusted), there is no need to perform any explicit actions to manage trusted certificates. In effect, the PKIBoot process allows a client to perform a remote query of trusted certificates from the CA user, implemented using standard PKI mechanisms. The process is illustrated in Figure 3. In this case the cryptlib PKIBoot user is acting as a proxy for the cryptlib CA user, however it can also act as a proxy for another cryptlib PKIBoot user, for example when a departmental server relays certificates from an organisation-wide server.

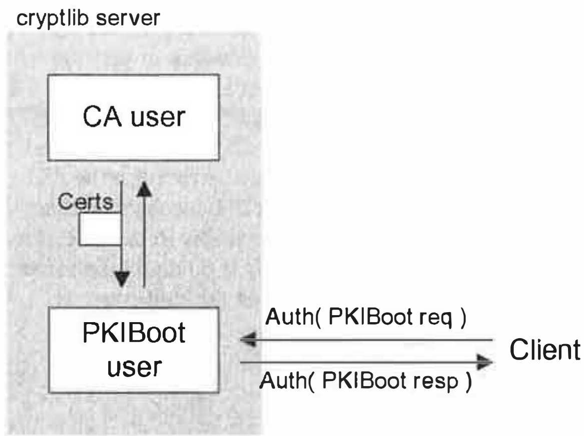


Figure 3: Role-based actions during the PKIBoot process

The list of trusted certificates is communicated in the standard CMS format accepted by any PKCS #7/CMS/-SMIME implementation [48], and by extension virtually every certificate-aware application. Although the transport protocol we're using is CMP, it can be in interpreted in a manner that allows the same bits-on-the-wire format as CMS (see section 5.3 for details on how this is possible), so that any certificate-using application can be PKIBoot-enabled through the use of a CMP front-end to handle the messaging.

There is a possible alternative format, Microsoft's Certificate Trust List (CTL) [50], but this only contains a hash of the certificate rather than the certificate itself (it's assumed that the certificates are distributed via some unspecified mechanism), contains additional unnecessary information such as policy identifiers alongside the hash, and most importantly contains a number of undocumented details and is apparently covered by a Microsoft patent. CMS predates CTLs by many years and is well-defined, widely-supported, and unencumbered by patents.

The entire process beyond defining the contents of the trusted certificate set is quite transparent to both the server administrator and the user, being handled automatically by the (modified) CMP server and cryptlib's role-based access control system. From the users' point of view, the PKIBoot process is simply a part of the initial certificate exchange and doesn't require any explicit actions or effort, being protected by the security mechanisms used in the initial certificate exchange (see the next section).

4.3 CMP

The final stage of the process consists of obtaining the user's certificate (or subsequent certificates). At this point the software (or hardware, if a crypto token such as a smart card is being used) generates a public/private key pair for the user and uses it to obtain the initial certificate. This is the first point at which the user needs to perform any explicit actions, namely entering their user name and password, just as they'd be required to enter this information when signing on to an ISP account or connecting to a corporate network. The same information was also used to authenticate the PKIBoot exchange, but as mentioned in the previous section this occurs as a transparent addition to the CMP process, so the user isn't required to provide additional information for the PKIBoot process or even know that it's occurring, in much the same way that DHCP sets things up for IP operation. There may also be other supplementary actions required at this point, for example if the CA charges for certificates issued then the user should be required to acknowledge that the operation that follows will result in some form of monetary charge being applied.

This process, while in theory probably the simplest part of the plug-and-play PKI operation, is in fact the most awkward, being considerably hampered by the extraordinary complexity and ambiguity of the CMP protocol specification. The problems encountered are covered in more detail in the discussion of implementation issues in section 5.2, with resolutions provided in section 5.3.

5 Discussion

The section examines some of the major issues that were encountered in implementing the operations covered in section 4, and concludes with a real-world acid test of the resulting system that examines whether it can live up to the claim of being a PKI your mother can use.

5.1 Service Location

In practice the use of SLP, while no doubt the correct thing to do, is somewhat impractical. An informal survey of sysadmins indicated that they would be reluctant to have yet another network service running on their servers, not helped by the fact that there were easier alternatives available. In the end, the use of HTTP-based mechanisms prevailed.

5.2 Problems with CMP

CMP is an extremely complex protocol with a large number of ambiguous, redundant, or even incomprehensible features (there were some protocol features that were removed at one stage when even the RFC authors couldn't explain their purpose). In many cases the function of various protocol elements is

unknown, and so implementors have to guess at their purpose. Needless to say, everyone guesses differently. For example, there are three different fields that function as nonces, with no-one certain as to their exact purpose or use (one interop report states that "no realistic replay attack was identified, it is not clear if nonces are required in [message types]" [51]). As a result, some implementations implement them, some don't, and some ignore them if they find them.

There is another nonce field described as a transaction ID that appears to be the most obvious place for a nonce, however some implementations use this field to identify the sender of the message. Others omit it (it's optional) and use the sender key ID field (also optional, and omitted by some implementations) to identify the sender. Still others use the sender X.500 Distinguished Name (DN) field for identification. This is the only mandatory ID field, but when obtaining their first certificate the user doesn't know their DN yet (in fact most users don't know their DN at any point, even when they have a certificate), so the field must be filled with a dummy value, which is rejected by some servers that (somehow) require the user to specify a DN in the initial request. Other servers accept any arbitrary DN, but return a certificate with a totally different DN of their own choosing, leaving the reason for needing to supply a DN in the first place open to speculation. To add to the confusion, there is a second DN present in the request itself that may or may not match the sender DN, and the sender DN may itself be overridden by the sender key ID, if present.

If pure DN-based identification is used, it's not possible to uniquely identify the certificate to be used to authenticate a CMP transaction, since a DN can only identify the overall owner of a group of certificates, but not an individual certificate. This problem was demonstrated by one CMP CA who updated their CA certificate, leading to the existence of multiple certificates identified by the same DN, so that all signatures created by the CA mysteriously failed to verify, with no easily discernable reason (this is purely a fault of the CMP design, existing data formats such as CMS/PKCS #7 don't exhibit any of these problems).

Further complications abound. Many fields are marked as mandatory, but never used, so special values have to be invented to fill them, with no other meaning than to indicate that their presence has no meaning. Other fields are marked as optional but are in fact mandatory, causing the protocol to break if they are omitted (because of the confusion over the purposes of many of the fields, it's not easy to determine this in advance). There are portions of CMP that serve no identifiable purpose (for example "what is the purpose of [part of protocol]? When should it be checked? What attack(s) does it protect against? Should it always be there or is it used in lieu of other mechanisms?" [51]), with the result that implementations omit them in order to reduce complexity, leading to lively debates over standards-compliance when a new implementation that does support the feature appears (during one interop session, interoperability among implementations was actually *worse* than it had been at the previous year's interop). Some protocol features were in fact moved from mandatory-to-support to optional when it was found that no-one had bothered to implement them.

In summary, trying to work with CMP was without doubt the most frustrating part of implementing the plug-n-play PKI design. Although it's easy enough to manage in a homogeneous environment, trying to achieve interoperability among multiple different implementations is at best difficult and at worst impossible, for example when one implementation requires that an X.500 DN be provided before the user knows what it is. The only mitigating factor here is that any CMP implementation configured to use the PKIBoot process will presumably be able to handle the necessary exchange of messages, leading to a de facto PKIBoot-capable common interpretation of the CMP specification.

5.3 Fixing CMP

In order to increase the chances of having two CMP implementations able to interoperate, we use an interpretation of the protocol designed to minimise problems. To work around the confusion involving identifiers, we add an extension containing an ESSCertID (a universal, unambiguous certificate identifier containing a hash of the certificate in question) [52] to all messages. This is possible because CMP uses a peculiar defunct form of ASN.1 which is imprecise enough that we can embrace and extend the protocol to use this type of identifier. This also allows us to use CMP to communicate standard PKCS #7/CMS certificate chains usable by any other certificate-using software as described in section 4.2, even if it doesn't talk CMP.

The use of the ESSCertID does away with the need to work with the unreliable CMP identification mechanisms. If there's no ESSCertID present, we take guesses at the various identifiers mentioned in section 5.2, trying them as key identifiers, user identifiers, and X.509 subject key identifiers (an X.509 certificate extension type). The X.500 DNs, which may be present, present but set to a special value to indicate that they're not present, or arbitrarily changed by the other side, are ignored since they're more trouble than they're worth.

The optional fields labelled as nonces are copied across into replies in case the other side pays attention to them, but otherwise ignored in order to interoperate with implementations that don't use them. The main nonce, labelled the transaction ID (although some implementations overload it as a user ID) is treated as the principal nonce, unless it's absent, in which case we fall back to the other nonces if they are present (as this description implies, the processing logic for CMP messages tends to be quite complex). Because of the uncertainty about nonces, the cryptlib CMP implementation is designed to function without them if necessary, employing a database with full transactional capabilities to record every operation so that an attempt to replay (for example) a certificate issue request will be detected by the presence of a duplicate entry in the database [53].

In practice then we use the ESSCertID to identify the certificate or key used for authentication of messages and the transaction ID (with a failsafe in the transaction processing system) as the nonce, falling back through a variety of alternatives if one or both are missing. If a certain minimum standard isn't reached (for example if all of the various nonce fields are absent, or none of the various identifier fields appears to identify a certificate or key), we abort the processing. As with the assumption about PKIBoot capabilities in section 5.2, it's safe to assume that anything capable of handling PKIBoot will also handle identifiers and nonces in a sensible manner.

5.4 Testing

The final step in the process was to test the implementation to determine whether this really was "a PKI your mother can use" (the author resisted the temptation to cheat slightly and run the client in baby-duck mode, which would have allowed him to claim that he had a PKI his potted plants could use). The test system consisted of a Windows '98 home PC, with an HTTP redirect leading to a PKIBoot/CMP server located on the other side of the world, partially to prove that remote certification services were just as feasible as local ones, and partially just because it was convenient.

Taking a recently-calibrated reference mother, the system was stress-tested to determine usability by non-technical users. The whole process functioned exactly as expected, with the end result being a freshly-minted certificate stored in the system without the user being aware that it had happened. The implementation actually obtains two certificates, first a long-term signature certificate authenticated with a MAC derived from the user name and password, and then a short-term encryption certificate authenticated with the just-obtained signature certificate, providing the necessary separation of encryption and signature keys which is so often sacrificed for ease-of-use when obtaining certificates using conventional mechanisms. This process is totally transparent to the user, as is the fact (or even the need to know) that two different keys are in use. A similar process is used in OpenPGP [54], which maintains Elgamal encryption keys authenticated with DSA signing keys.

A second test was used to verify functionality when run in baby-duck mode, with an old headless Linux box acting as a stand-in for the Internet-enabled toaster. This test was pretty much a tautology: the system went through the PKIBoot and certificate-acquisition process, and subsequently initiated an SSL connection authenticated from the server side with a trusted certificate exchanged during the PKIBoot process and from the client side with its newly-obtained certificate. In the real world, this would presumably be followed by the downloading of control software or configuration data over the SSL-secured link, or in the future possibly via a dedicated protocol [44]. In contrast in a world without plug-and-play PKI services, SSL server authentication is performed using a hardcoded arbitrary collection of mostly-unknown CA certificates, and client authentication is omitted because it's just too hard to manage.

In attempting to design a security protocol with usability as a major goal, it's not possible (without access to the usability testing labs of a large software company) to immediately determine whether this goal has indeed been met. The intent of this paper was to create a design that makes the process as simple as possible. A more general test than the previous two, in the form of ongoing use in cryptlib by general users, is currently under way. To some extent the design has been vindicated by the fact that at least one beta-tester wanted to move the initial experimental implementation into production immediately, but in the long run only time will tell how successful the overall design is.

In conclusion, the system passed the initial tests, providing a fully automated means of acquiring (and later updating/replacing) certificates usable by even the most inexperienced user, or as part of an automated process requiring no user intervention. In addition, the system enforced best-practice key usage without the user having to be aware of this — they weren't even aware that they were using certificates, and were quite amazed when informed that this had achieved significantly more than what was achieved by the tortuous process covered in section 1.

The results of this work are, as always, available online [6].

6 Conclusion

This paper has examined the design process involved in making the various steps of establishing and using a PKI simple and automated enough that anyone can use it, including embedded devices that can't rely on manual intervention by a human to provide them with the necessary configuration. What appears at first glance to be a relatively simple concept is complicated by the lack of established standard(s) to help accomplish the task, and the high level of difficulty encountered in working with the standards created to allow certificate management operations. In contrast, the initial part of the PKI setup operation suffers from an embarrassment of riches, with the eventual choice of mechanism being driven by ease-of-use and ease-of-deployment considerations even if it may not be the most appropriate on purely technical grounds. Despite these difficulties, it proved possible to use (and in some cases slightly abuse) existing standards to accomplish the task at hand, without requiring the creation of yet another set of PKI standards to perform the task, with important portions of the protocol output (for example PKCS #7/CMS certificate chains sent over CMP) directly usable by existing applications.

Acknowledgements

The author would like to thank cryptlib users for their feedback on plug-and-play PKI requirements, his mother for acting as a guinea pig during testing, and the anonymous referees for their helpful comments on the paper.

References

[1] "Peer to Peer: Collaboration and Sharing over the Internet", Bo Leuf, Addison-Wesley, 2002.

[2] "Network Security with OpenSSL", John Viega, Matt Messier, and Pravir Chandra, O'Reilly and Associates, June 2002.

[3] "OpenSSL: The Open Source toolkit for SSL/TLS", `http://www.openssl.org`.

[4] "Re: Purpose of PEM string", Doug Porter, posting to pem-dev mailing list, message-ID `93Aug16.003350pdt.13997-2@well.sf.ca.us`, 16 August 1993

[5] "The Design of a Cryptographic Security Architecture", Peter Gutmann, *Proceedings of the 8th Usenix Security Symposium*, August 1999, p.153.

[6] "cryptlib Encryption Toolkit", `http://www.cs.auckland.ac.nz/-~pgut001/cryptlib/index.html`.

[7] Lucky Green, private communications.

[8] David Harris, private communications.

[9] "Gateway Guardians", Fred Avolio, *Information Security*, **Vol.6**, **No.2** (February 2003), p.51.

[10] "1st Annual PKI Research Workshop", `http://www.cs.dartmouth.edu/~pki02/`.

[11] "Amusing note on real-world PKI deployment", Peter Gutmann, posting to the cryptography@wasabisystems.com mailing list, message-ID `200201251007.XAA283003@-ruru.cs.auckland.ac.nz`, 25 January 2002.

[12] "Computer Communications Security: Principles, Standard Protocols and Techniques", Warwick Ford, Prentice-Hall, 1994.

[13] "Network Security: Private Communication in a Public World (2nd ed)", Charlie Kaufman, Radia Perlman, and Mike Speciner, Prentice-Hall, 2002.

[14] "Security in Computing (3rd ed)", Charles Pfleeger, Shari Lawrence Pfleeger, and Willis Ware, Prentice-Hall, 2002.

[15] "Lessons Learned in Implementing and Deploying Crypto Software", Peter Gutmann, *Proceedings of the 11th Usenix Security Symposium*, August 2002, p.315.

[16] "The Crypto Gardening Guide and Planting Tips", Peter Gutmann, January 2003, `http://www.cs.auckland.ac.nz/-~pgut001/pubs/crypto_guide.txt`.

[17] "Good-Enough Security: Toward a Pragmatic Business-Driver Discipline", Ravi Sandhu, *IEEE Internet Computing*, **Vol.7**, **No.1** (January/February 2003), p.66.

[18] "Online Authentication", `http://www.e-government.govt.nz/authentication/-index.asp`.

[19] "Zero-configuration Networking", Eric Guttman, *Proceedings of INET 2000*, July 2000.

[20] "Autoconfiguration for IP Networking: Enabling Local Communication", Eric Guttman, *IEEE Internet Computing*, **Vol.5**, **No.3** (May/June 2001), p.81.

[21] "Zero Configuration Networking (zeroconf)", `http://www.ietf.org/html.charters/-zeroconf-charter.html`.

[22] "RE: IP: SSL Certificate "Monopoly" Bears Financial Fruit", Lucky Green, posting to the cryptography@wasabisystems.com mailing list, message-ID `002901c228b4$14522fb0$-6501a8c0@LUCKYVAIO`, 11 July 2002.

[23] "A rant about SSL, oder: die grosse Sicherheitsillusion", Matthias Bruestle, presentation at the KNF-Kongress 2002.

[24] "Internet X.509 Public Key Infrastructure: Logotypes in X.509 certificates", Stefan Santesson, Russell Housley, and Trevor Freeman, IETF draft, March 2003.

[25] "Dynamic Host Configuration Protocol", RFC 2131, Ralph Droms, March 1997.

[26] "Core Jini (2nd ed)", W.Keith Edwards, Prentice-Hall, 2000.

[27] "The Jini Specification (2nd ed)", Ken Arnold (ed), Addison-Wesley, 2000.

[28] "Universal Plug and Play Specifications", `http://www.upnp.org/resources/-specifications.asp`.

[29] "UPnP Design by Example: A Software Developer's Guide to Universal Plug and Play", Michael Jeronimo and Jack Weast, Intel Press, 2003.

[30] "Simple Object Access Protocol (SOAP) 1.1", W3C Note, 8 May 2000, `http://www.w3.org/TR/SOAP/`.

[31] "Service Location Protocol, version 2", RFC 2608, Erik Guttman, Charles Perkins, John Veizades, and Michael Day, June 1999.

[32] "Service Templates and Schemes", RFC 2609, Erik Guttman, Charles Perkins, and James Kempf, June 1999.

[33] "An API for Service Location", RFC 2614, James Kempf and Erik Guttman, June 1999.

[34] "Service Location Protocol: Automatic Discovery of IP Network Services", Erik Guttman, *IEEE Internet Computing*, **Vol.3**, **No.4** (July-August 1999), p.91.

[35] "OpenSLP", `http://www.openslp.org`.

[36] "Salutation Consortium", `http://www.salutation.org`.

[37] "Rendezvous", `http://developer.apple.com/macosx/-rendezvous/`.

[38] "Salutation and SLP", Pete St. Pierre and Tohru Mori, `http://www.salutation.org/techtalk/-slp.htm`.

[39] "A DNS RR for specifying the location of services (DNS SRV)", RFC 2782, Arnt Gulbrandsen, Paul Vixie, and Levon Esibov, February 2000.

[40] "Internet X.509 Public Key Infrastructure Operational Protocols: Certificate Store Access via HTTP", Peter Gutmann, IETF draft, March 2003.

[41] "Re: Computation of issuerKeyHash in OCSP", Peter Gutmann, posting to the ietf-pkix@imc.org mailing list, 13 March 2001.

[42] "The Resurrecting Duckling: Security Issues in Ad-Hoc Wireless Networking", Frank Stajano and Ross Anderson, *Proceedings of the 7th International Workshop on Security Protocols*, Springer-Verlag Lecture Notes in Computer Science No.1796, April 2000, p.172.

[43] "The Resurrecting Duckling — What Next?", Frank Stajano, *Proceedings of the 8th International Workshop on Security Protocols*, Springer-Verlag Lecture Notes in Computer Science No.2133, April 2000, p.204.

[44] "Using CMS to Protect Firmware Packages", Russ Housley, IETF draft, April 2003.

[45] "Internet X.509 Public Key Infrastructure: Certificate Management Protocols", RFC 2510, Carlisle Adams and Stephen Farrell, March 1999.

[46] "Certificate Management Messages over CMS", RFC 2797, Michael Myers, Xiaoyi Liu, Jim Schaad, and Jeff Weinstein, April 2000.

[47] "Planning for PKI: Best Practices Guide for Deploying Public Key Infrastructure", Russ Housley and Tim Polk, John Wiley and Sons, 2001.

[48] "Cryptographic Message Syntax (CMS)", RFC 3369, Russell Housley, August 2002.

[49] "DHCP Options for Service Location Protocol", RFC 2610, Charles Perkins and Erik Guttman, June 1999.

[50] "Certificate Trust Lists: What Are They? Why Are They Useful?", Trevor Freeman, presentation at the NIST PKI Working Group meeting, 12 November 1998.

[51] "CMP Interoperability Testing: Results and Agreements", Robert Moskowitz, IETF draft, June 1999.

[52] "Enhanced Security Services for S/MIME", RFC 2634, Paul Hoffman, June 1999.

[53] "A Reliable, Scalable General-Purpose Certificate Store", Peter Gutmann, *Proceedings of the 16th Annual Computer Security Applications Conference (ACSAC'00)*, December 2000, p.278.

[54] "OpenPGP Message Format", RFC 2440, Jon Callas, Lutz Donnerhacke, Hal Finney, and Rodney Thayer, November 1998.

Analyzing Integrity Protection in the SELinux Example Policy

Trent Jaeger Reiner Sailer Xiaolan Zhang
IBM T. J. Watson Research Center
Hawthorne, NY 10532 USA
Email: {jaegert,sailer,cxzhang}@us.ibm.com

Abstract

In this paper, we present an approach for analyzing the integrity protection in the SELinux example policy. The SELinux example policy is intended as an example from which administrators customize to create a policy for their site's security goals, but the complexity of the model and size of the policy make this quite complex. Our aim is to provide an access control model to express site security goals and resolve them against the SELinux policy. Ultimately, we aim to define a minimal trusted computing base (TCB) that satisfies Clark-Wilson integrity, by first testing for the more restrictive Biba integrity policy and resolving conflicts using Clark-Wilson semantics. Our policy analysis tool, Gokyo, implements the following approach: (1) it represents the SELinux example policy and our integrity goals; (2) it identifies conflicts between them; (3) it estimates the resolutions to these conflicts; and (4) provides information for deciding upon a resolution. Using Gokyo, we derive a proposal for a minimal TCB for SELinux includes 30 subject types, and we identify the work remaining to ensure that TCB is integrity-protected. Our analysis is performed on the SELinux example policy for Linux 2.4.19.

1 Introduction

A goal for many years has been effective mandatory access control (MAC) for UNIX systems. By an *effective* MAC system, we envision that system administrators can define access control policies that guarantee site security goals while enabling the convenient execution of applications. Early MAC policies, such as the Bell-LaPadula secrecy policy [2] and the Biba integrity policy [4], defined clear security goals, but were too restrictive for convenient use for UNIX applications. Commercial operating systems that were extended to meet Orange Book B1 (i.e., MAC plus other features) were not broadly applied (i.e., mainly aimed at government installations). Recent efforts at MAC systems use flexible access control models to achieve convenient use (e.g., DTOS, Flask [17, 21], etc.), but demonstrating that particular security goals have been met is more difficult (and these systems have not been widely used either). Flexible access control models typically result in more complex policies, so it is more difficult to determine if these policies have the desired effect.

The recent addition of the Linux Security Modules (LSM) framework [22] enables the MAC enforcement for the Linux kernel. The LSM framework is designed to be agnostic to the MAC approach, and it has been designed to support modules with flexible MAC models. The most comprehensive and flexible module for LSM is the SELinux module [18]. While SELinux supports a variety of policy models itself, an extended Type Enforcement (TE) model [5] is used for most policy development. An example policy is under development that consists of a set of UNIX service and application policies that each aim to ensure effective operation while preventing security vulnerabilities. The example policy does not define a secure system, but serves as a basis for developing a secure system once the security goals are defined. The extended TE model is rather complex (i.e., consists of a large number of concepts) and the SELinux example policy is large (e.g., 50,000+ policy statements in the `policy.conf` for Linux 2.4.19), so customization of the SELinux example policy to a policy that guarantees satisfaction of system security goals is an arduous and error-prone task.

While the use of a simpler access control model might make it easier to ensure that security goals are met, we believe that this would result in applications failing to run conveniently, and ultimately, the circumvention of these security goals. The comprehensive nature of the SELinux policy model enables flexible trade-off between application and security goals. For example, the SELinux example policy itself is developed by proposing application policies and refining them based on the policy violations that may be generated. Thus,

the SELinux example policy itself is a direct result of making these trade-offs.

The question is whether a manageable set of effective security goals can be described and verified for SELinux policies. Obviously, it is highly unlikely that the SELinux example policy adheres to a simple high-level policy, such as the two-level integrity model of LOMAC [9]. However, the policy may be sufficiently close to such a policy that the conflicts can be managed (i.e., either a small number or a small number of equivalence classes). If so, then verification may be possible by verifying the general goals and using ad hoc techniques to resolve the conflicts. We have found that this approach holds some promise for application policies, in particular the Apache administrator [12], but do not know whether this can work for the trusted computing base (TCB) subjects in the SELinux policy. Obviously, if we cannot prove that the TCB is integrity-protected, its system cannot be considered secure.

In this paper, we propose a near-minimal TCB for SELinux systems and examine how to verify that this TCB is integrity-protected. First, we define integrity relationships between the TCB subject types and less trusted system and application subject types. Second, we input these constraints into our policy analysis tool, called Gokyo [12], and identify integrity conflicts between the TCB and the system. The Gokyo tool enables flexible expression of conflict sets and their resolution, so our next goal is to determine what resolutions appear feasible for TCB integrity conflicts. Using Gokyo, we classify conflicts into classes based on their likely resolution. Since most resolutions depend on ad hoc information, it is still a manual process to complete the analysis. Using Gokyo, we identify a minimal TCB for the SELinux example policy of 30 subject types, half of which are infrequently-used administration subjects. To use this TCB, 5 sanitization problems must be solved, but we believe that most can be addressed in practice, including the use of Gokyo itself to manage the broad file access rights currently granted to trusted subjects. Ultimately, Gokyo is useful in identifying problems in meeting security goals, classifying these problems, and providing information for resolving them.

The paper is structured as follows. In Section 2, we examine the SELinux extended Type Enforcement model and outline our site security goals for that model, integrity protection of a minimal trusted computing base. In Section 3 we describe our approach to resolving a policy against our integrity protection requirements. In Section 4, we detail the implementation of our analysis using Gokyo and present our analysis results. In Section 5, we present related work, and we conclude in Section 6.

2 SELinux Security Goals

2.1 SELinux Policy Model

While SELinux supports a variety of access control policy models [21], the main focus of SELinux policy development has been an extended Type Enforcement (TE) model [1, 5, 20]. In this section, we provide a brief overview of the SELinux policy model concepts, focusing only on the concepts that are relevant to the analysis that we perform. A number of other concepts are represented in the SELinux extended TE model, such as roles and identity descriptors, that we do not cover here. A detailed description of the SELinux policy model is given elsewhere [20].

The traditional TE model has subject types (e.g., processes) and object types (e.g., files, sockets, etc.), and access control is represented by the permissions of the subject types to the object types. In SELinux, the distinction between subject and object types has been dropped, so there is only one set of types that are object types and may also act as subject types.

The SELinux extended TE model is shown in Figure 1. All objects are labeled with a *type*. All objects are an instance of a particular *class* (i.e., data type) which has its own set of *operations*. A permission associates a type, a class, and an operation set (a subset of the class's operations). Thus, permissions associated with SELinux types can be applied independently to different classes. For example, different rights can be granted to a user's files than to their directories. In fact, since the objects are of different classes, they have different operations. Should the administrator want to give different access rights to two objects of the same class, then these objects must belong to different types.

Permission for a (subject) type to perform operations on a(n) (object) type are granted by the *allow* statement. Any element of the permission relationship can be expressed using this statement, so the expression of least privilege rights is possible. The *dontaudit* statement provides a variation on the basic permission assignment. A combination of allow statements result in a union of the rights specified, whereas a combination of dontaudit statements on the same type pair and class are intersected.

In addition, the extended TE model also has type at-

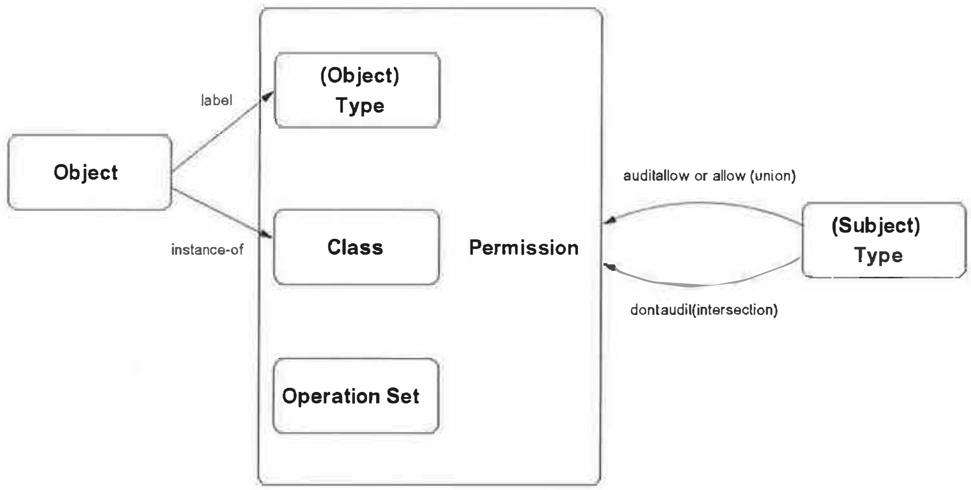


Figure 1: SELinux extended Type Enforcement (TE) policy model basics.

tributes that represent a set of types (i.e., all the types with that attribute assigned). Type attributes enable assignment to multiple types at a time. For example, a permission can be assigned to each subject type with that attribute or a subject can be assigned permission to each object type with that attribute.

Containment is enforced by limiting the permissions accessible to a subject type (as described above), limiting the relabeling of object types, and limiting the domain transitions that can be made by a subject type. Relabel rights are controlled in SELinux by limiting access to *relabelfrom* and *relabelto* operations. As the names indicate, *relabelto* enables objects to be relabeled to that type and *relabelfrom* enables objects of a particular type to be relabeled.

Domain transitions can occur when a subject type executes a new program. Again, SELinux defines an operation, called *transition*, to perform these transitions. A subject type must have a transition permission for the resultant subject type in order to affect a domain transition.

The SELinux model also has statements for *type transition* and *type change*. Type transition statements are used by SELinux to automatically compute transitions, but are not necessary for control (i.e., transition permissions are always necessary). Type change statements alter the type of an object upon access by the specified subject type. Such statements are useful when a system administrator logins in using a user's tty. Type change statements transition the object type of the tty to prevent users from altering input.

In order to simplify the task of expressing policies, the SELinux extended TE model also includes a large number of macros for expressing sets of policy statements that commonly occur together. We do not examine the policy macros in detail because policy analysis requires us to understand the policy at the level of the type enforcement model statements (i.e., which subject types can perform which operations on which object types).

2.2 SELinux Example Policy

The SELinux community is working jointly on the development of UNIX application policies whose composition is called the *SELinux example policy*. The SELinux example policy does not define a secure system, but is intended as input to the development of a custom policy for each site's security goals, commonly called a *security target*. Unfortunately, customization is not simply composition of the policies for the applications of interest. The application policies themselves are somewhat specialized to the environment in which they were developed, and interactions between the policies of multiple applications may lead to vulnerabilities. In general, the composition of policies that are proven secure may not result in a secure system.

The task of customization is further complicated by the size of the example policy and the complexity of the extended TE model described in Section 2.1. The SELinux example policy for Linux 2.4.19 consists of over 50,000 policy statements (i.e., the processed macro statements in `policy.conf`). According to our analysis, this specification represents over 700 subject types and 100,000 permission assignments. We believe that size and complexity of the SELinux example policy

make it impractical to expect that typical administrators can customize it to ensure protection of their trusted computing base (TCB) and to satisfy their site's security goals on this TCB. This may seem obvious to some and may seem insufficiently justified to others, but we will describe a more detailed argument on why we believe this in Section 2.3.

Despite this, we are convinced that the SELinux example policy is valuable to building secure systems, for these two reasons primarily: (1) it provides a flexible enough representation to capture the permissions necessary for UNIX applications to execute conveniently and (2) it provides a comprehensive definition of a reference monitor for UNIX. First, the SELinux example policy is developed per application in a manner that identifies a superset of the permissions required to run an application conveniently while possibly meeting a particular security target. What typically happens is that a proposal is made for an application policy, then this policy is tested by the community when they use the application. Since SELinux reports authorization failures (i.e., the lack of a permission requested), it is much easier to determine that insufficient permissions were assigned than whether a security vulnerability is created. Thus, a verified proposal for least privilege permissions for each application is represented by the SELinux policy. What we need is a better way to test whether our security goals are satisfied, such that conflicts can be identified and addressed.

Second, the SELinux example policy is a comprehensive representation of UNIX access control. The SELinux model aims to comprehensively control access to all classes (i.e., kernel data types) that may be operated upon by a user-level Linux process. There are 29 classes defined in the SELinux example policy. Each class has its own set of operations that are intended to capture all the relevant subtleties in accessing and modifying a class. Given the scope of the SELinux example policy at this granularity, the SELinux example policy provides as precise and comprehensive a repository of UNIX application access control information as exists today. We need to leverage this repository in the development and refinement of security goals, but provide such leverage through higher-level concepts that enable effective management.

2.3 SELinux Security

Unlike early MAC models like Bell-LaPadula [2] and Biba [4], a TE model does not explicitly indicate the security goals of the policy. Thus, the policy implies the security goals of the system. For a TE system, more like an access matrix, we only learn that certain subjects can only perform certain operations on certain objects. The security goals of the policy are not represented at a higher-level than this.

The SELinux model provides an approach by which secrecy and integrity properties may be achieved with least privilege permissions and containment of services [16]. The system administrators create a policy that is restrictive with respect to granting rights that violate secrecy and integrity properties and we use the notions of least privilege and containment to minimize the damage due to compromises where these occur.

From our perspective, the integrity of the TCB is the basis of security, so that is the focus of our analysis. In general, it is preferable to have a "minimal" TCB. The smaller the TCB, the easier it is to verify the components. However, if the minimal TCB subjects are dependent on other subjects, then these other subjects must be added to the TCB or dependencies must be removed. In this paper, we will identify dependencies and determine how to resolve them to keep our TCB as small as is feasible.

Since we are striving for a minimal TCB, we do not assume a two-level integrity system (system and user) as in LOMAC [9], but rather we start with the most fundamental system services and try to determine how the integrity of these can be enforced. Applications may further depend on other subjects. For example, Apache depends on other system services and the Apache administrator. We believe these are at two distinct integrity levels [13]. In this paper, we examine only explicitly examine the TCB and non-TCB boundary.

Further, we note that the benefits of least privilege permissions and containment are not relevant to the protection of the TCB. Since the TCB subject types can legitimately transition to any other subject type, containment is not possible for the TCB subjects. Therefore, the focus is on the integrity of these services.

Figure 2 shows the SELinux example policy's type transition hierarchy for our proposed TCB subject types [1]. `kernel_t` is the primordial subject type in the SELinux system. It transitions to `init_t` which then can start a variety of services. Key to our analysis are the administrative (e.g., `sysadm_t`, `load_policy`, `setfiles_t`, etc.) and authentication subject types (e.g., `sshd_t`, `local_login_t`, etc.) that determine the basis for security decisions in SELinux. We also in-

[1] This hierarchy is generated by the *transition* permissions held by each of these subject types.

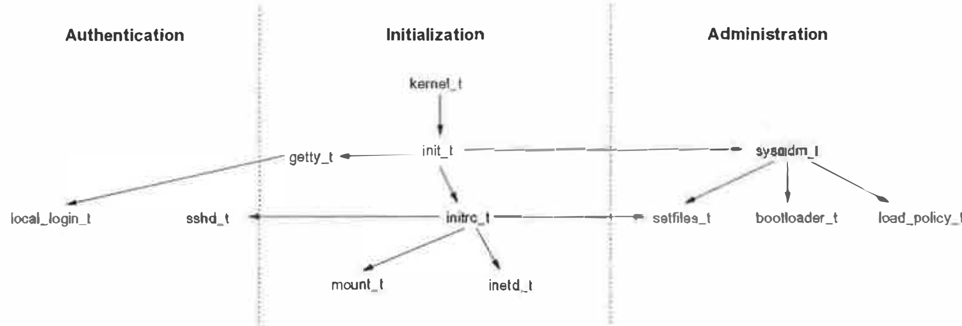


Figure 2: SELinux Example Policy's type transition hierarchy for our proposed TCB subject types.

clude `initrc_t` and `inetd_t` because these services initiate many of the services in a UNIX system.

Of course, there are lots of other services upon which the correct execution of applications is necessary (over 80 identified for the Apache administrator [13]), but we chose this proposal for a minimal TCB based primarily on the early appearance of these services in the type transition hierarchy and the amount of transition "fan-out." Both of these features indicate that vulnerabilities in that subject type will be difficult to contain.

While this TCB represents a small number of subject types, the complexity of their interactions with the rest of the system in the SELinux policy makes manual verification impractical. First, each subject type is included in around 500 to over 1000 policy statements in `policy.conf`. Manual examination of this many statements alone is impractical, but these statements must be compared to the other thousands to determine whether a significant conflict exists. Automated tools are necessary to represent the security goals, identify conflicts, and provide as much support as possible to conflict resolution.

2.4 Integrity Requirements

The goal of our analysis is to protect the integrity of our trusted computing base (TCB), so we need to define more precisely what we mean by this. Traditional policies for integrity protection include Biba [4] and Clark-Wilson [6]. The Biba integrity property is fulfilled if all the higher-integrity processes do not depend on lower-integrity processes in any manner. This implies that a high integrity process cannot read lower-integrity data, execute lower-integrity programs, or otherwise obtain lower-integrity data in any other manner. As a result, a process cannot write data above its integrity level. Therefore, if high and low integrity processes write to the same file, then it must be a low integrity file. Obviously, the high integrity process can no longer read from this file and maintain Biba integrity.

Of course, UNIX applications do not obey a strict Biba integrity. Often higher-integrity processes read input generated by lower integrity processes, but in many cases it is assumed that they can handle this input. When they cannot, we often identify this as a vulnerability in the supposedly high-integrity program [2]. The Clark-Wilson integrity model attempts to capture this notion. In the Clark-Wilson model, *constrained data items* (CDIs) are high-integrity data that are processed only by certified *transformation procedures* (TPs). However, TPs may also process *unconstrained data items* (UDIs). This is similar to high integrity programs processing low integrity data. The Clark-Wilson model also includes *integrity verification procedures* (IVPs) that can be used to verify the integrity of CDIs at particular times.

The Clark-Wilson model is based partly on certification of the components (IVPs and TPs) and partly on their enforcement of particular properties. We do not address certification here, but we examine the plausibility of meeting Clark-Wilson enforcement requirements using SELinux (paraphrased from the Clark-Wilson paper [6]):

- **E1:** Each TP operates on a particular list of CDIs and CDIs are only manipulated by a TP.
- **E2:** The system must maintain a list of subjects, TPs, and the CDIs those TPs may reference, and only those references are permitted.
- **E3:** The system must authenticate the identity of each user that attempts to execute a TP.

[2] Of course, the use of so many root services gives has given a false impression of the integrity of many programs.

- **E4:** The list of TPs and IVPs can only be changed by a subject permitted to certify those TPs and IVPs.

SELinux identifies object types, and thus, the objects manipulated by programs as stated in E1. However, it does not distinguish between CDIs and UDIs. Thus, we know whether CDIs are only processed by TPs, nor do we know which subjects are TPs, as required by E2. The satisfaction of E3 must be provided by the authentication infrastructure in a dependable manner (i.e., using TPs). The mandatory nature of SELinux policies implicitly enforce E4.

Thus, our task is to identify the TPs that define the SELinux example policy's TCB (to satisfy E2). Since we want to ensure the integrity of our TCB, the only CDIs are those processed by the TCB subjects. As a result, we only need to ensure the integrity of these. However, the TCB may operate on UDIs, so we need to distinguish between UDIs and CDIs. Thus, we will perform the following tasks: (1) propose TCB subjects; (2) identify possible low-integrity UDIs (i.e., data whose value may depend on some low-integrity subject); (3) determine whether these are true UDIs; and (4) resolve cases where we suspect that a CDI is processed by a non-TCB subject (i.e., a subject that is not executing a TP).

Since the identification of the use of low-integrity data is essentially Biba, we perform a Biba analysis on our proposed TCB relative to the SELinux example policy. We then perform analyses to classify possible UDIs based on the possible resolutions to the integrity issue.

2.5 Low-Integrity Data

We first distinguish between two types of dependencies on low-integrity data that violates Biba: (1) read integrity violations and (2) read-write integrity violations. The difference is that, in the second case, writes by our TCB may be revised by a lower integrity process. While this is not strictly an issue in Clark-Wilson (i.e., these data may be UDIs), we are not comfortable with the possibility that a TCB subject write UDI data. Thus, we always classify read-write integrity violations as likely CDIs.

If we believe data are likely to be CDIs, then we have the following options to resolve the conflicts: (1) trusting the low-integrity subject (i.e., add it to the TCB); (2) exclude the low-integrity subject; (3) exclude the conflicted object type; (4) policy override; and (5) change the policy. It is possible to extend the proposed TCB, but since we want to keep the TCB minimal and the addition of more subject types will probably introduce more constraints, this is a low priority option.

We can exclude either the conflicting object type or the low-integrity subject type from the system to resolve an integrity conflict. Since we are analyzing the SELinux example policy to create a security target, it is perfectly reasonable to remove subject that cause significant integrity issues that we do not trust. For example, `insmod_t` installs modules into the kernel, but for a high integrity system we will compile the modules into the kernel. Thus, integrity conflicts caused by this service can be ignored. The exclusion of object types may be less plausible given that the object may be necessary for correct processing, but there are some cases where this makes sense. For example, we can eliminate integrity conflicts if we preclude objects of the type `removable_device_t` which may be acceptable for a secure system.

Lastly, we can change the policy by adding overriding statements (e.g., denying the violating read or write) or by modifying the SELinux example policy itself. We have found that handling integrity violations as exceptions or defining special handling for conflicting assignments with common semantics are both effective in reducing the need to express even more complex and fine-grained policies [12]. Modifying the SELinux example policy is a last resort: it is complex enough.

If we believe data are likely to be UDIs, we may assume that the TP is protected or protects itself by sanitizing its UDI inputs. Certification may depend on receiving only specific inputs, in practice, so providing external sanitization may also be an option. We may also identify the need for other security processing, such as auditing and intrusion detection, upon use of UDIs. We see this simply as another alternative to denials.

3 Analysis Approach

The basic approach to evaluating the proposed TCB for the SELinux example policy is as follows. First, we identify Biba integrity violations between the TCB subject types and the rest of the SELinux example policy. Second, we try to classify our conflicts based on the concepts such as the type of integrity violation (i.e., read or read-write), the proposed integrity of the conflicting subject type (i.e., high or low), and the likelihood of exclusion (i.e., of object type or subject type). Third, we

perform some manual analysis to determine the likely solution and see if these results correlate with the classifications. This includes outlining implementations to support these classifications, particularly where sanitization or policy modification is the choice.

3.1 Gokyo Policy Analysis Tool

We have built a policy analysis tool called *Gokyo* that is designed to identify and enable resolution of conflicting policy specifications [12, 13]. The general concept that Gokyo enforces is *safety*. A policy specification is said to be *safe* if no subject can obtain an unauthorized permission [10]. If we take policy administration into account, then any permission can be assigned to any subject type by the administrator in a policy such as TE. Therefore, we need an additional specification concept, typically called *constraints*, to express the safety requirements on the administrators to ensure that our policy meets our goals.

Gokyo implements an approach called *access control spaces* whereby semantically meaningful permission sets are identified and handling can be associated with each set independently. While there are a variety of semantically meaningful permission sets, the most common to consider are: (1) those assigned to a subject type; (2) those precluded from a subject type by a constraint; and (3) the permissions whose assignment or preclusion status is unknown. The overlapping regions of these sets also form subspaces, so we can consider the set of permissions that are both assigned and precluded. Of course, these sets can be further decomposed to provide more effective control. For example, we may deny all integrity conflicts, except log writes, which we allow, and input from network objects, which we sanitize and audit.

The general idea is that by identifying semantics to the subspaces it may be possible to attach handling semantics with the entire subspace. This would permit administrators to keep the basic, simpler constraints that largely work and specify only the additional handling semantics. We have found that the Apache administrator policy for SELinux 2.4.16 largely adheres to a Biba integrity model, such that 19 conflicting cases can be handled as eight access control spaces [13].

Gokyo represents policies using a graphical access control model shown by example in Figure 3. Permissions (i.e., object types and the permitted operations), subject types, and subjects are represented by graph nodes. In addition to this information, permission nodes also store the *object class* (i.e., datatype) and *operations* permitted by the permission. Note that object types are represented by permissions with no rights. In general, a node represents a set, so it is possible to build set-hierarchies consisting of aggregations of individual permissions, subject types, and subjects.

Example 1 Figure 3 shows an example of an access control specification using this model. Subject $s1$ has values $S(s1) = s1$, $R(s1) = r2$, and $P(s1) = P(r2)$. That is, $s1$ represents one subject, $s1$, and is assigned to one subject type, $r2$. Since the only route from propagation of permissions is through $r2$, $s1$'s permissions are defined by $P(r2)$. The value of $P(r2) = P(p6)$ and, since $p6$ is an aggregate its permissions are $P(p6) = P(p4) \cup P(p5)$. Since $p5$ is an aggregate as well, its permissions can be further decomposed.

For expressing constraints in this model, we also use a set-based approach [11]. In general, constraints are expressed in terms of two sets and a comparator function, $set_1 \bowtie set_2$, where $\bowtie$ represents some comparator function. Such comparators are set operations, such as disjointness (i.e., null intersection), cardinality of intersection, subset relations, etc.

Example 2 We define a constraint type for *integrity*. An *integrity* constraint $x \parallel y$ where $x \in R \cup S$ and $y \in R \cup S$ means that the set of read and execute permissions of x must not refer to any objects to which y has write permissions.

For each subject type, Gokyo stores the assigned permissions and the prohibited permissions. The prohibited permissions are the permissions whose assignment to the subject would result in the violation of a constraint, so these permissions are represented in terms of the constraint [3]. Further, Gokyo identifies the access control space consisting of the intersection between the assigned and prohibited spaces. It is this space where conflict resolution is necessary.

3.2 Identifying Integrity Conflicts

Returning to the problem of analyzing our proposed TCB, the SELinux example policy represents the assigned permissions. We add Biba integrity constraints

[3] Details are beyond the scope of this paper. See the detailed Gokyo writeup [13].

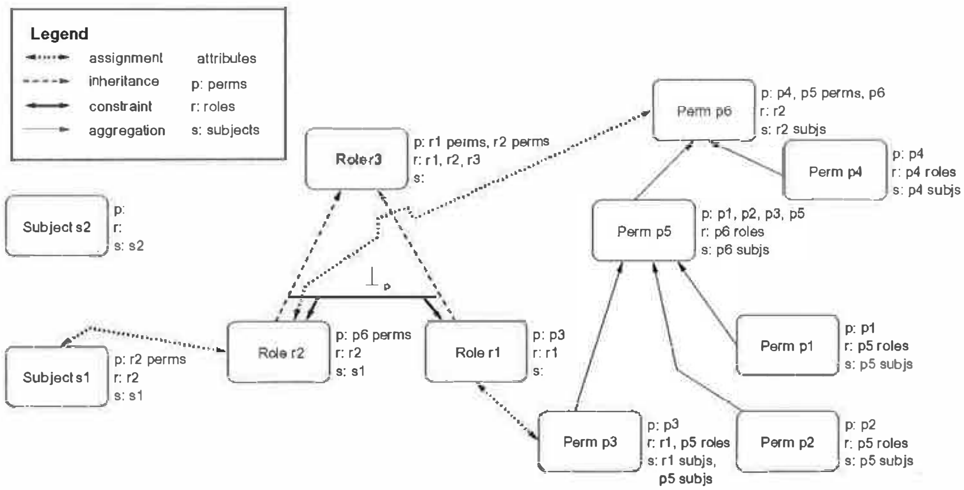


Figure 3: Example access control representation (the fields "p:", "r:", "s:" refer to the permissions, subject types, and subjects assigned to these entities, respectively)

between each of our TCB subject types and all other subject types in Gokyo. That is, for each TCB subject type, we add a constraint that it cannot apply a read/execute operation (i.e., an operation involving input of data to the subject type's processes) to any object type and class combination that is written by any other subject type in the system. Note that this constraint is more restrictive even than our original proposal, but understanding which subject types actually have integrity relationships may be useful in resolving conflicts.

Gokyo implements this constraint by computing read/execute permissions for each object type and class combination written by the other subject types. This set of read permissions is the set of permissions that the TCB subject type may not read. When the constraint is tested, if the TCB subject type has a read/execute permission that intersects one of the precluded permissions then a constraint violation is generated. In some cases, a single allow statement may result in several constraint violations. This can occur when an allow is made on a type attribute rather than a type directly. For example, access to read a variety of network input is the result of a single allow statement. Such assignments are propagated to each object type that has this type attribute. Gokyo shows only the unique assignments that result in violations, but can print all the individual violations to a file. In our results, we will also focus on the unique assignment mainly.

3.3 Classifying Conflicts

Once the conflicting subspace for a TCB subject type is generated, we could choose a handler for this subspace. In general, Gokyo permits overriding the conflict by granting or denying the subspace. However, both grants and denials can be augmented by arbitrary analysis code ranging from audit to complex intrusion detection. Thus, if the integrity conflicts that we find are all representative of the same situation we could choose a single approach to handling them.

In Section 2.4 lists five approaches for dealing with integrity conflicts that are summarized in Table 1. The problem is to determine which integrity conflicts imply which resolutions; the SELinux example policy does not provide any further input explicitly. To address this we classify conflicts in a manner that does not unequivocally identify the resolution, but does identify the possible resolutions. First, members of the TCB subject types may be trusted writers, so if the subject type of an integrity conflict is a TCB subject type then all handling options are possible. Further, some subject types may be candidates to be added to the TCB. Subject types with a significant number of conflicts should be considered. We use the heuristic that subject types with an average of greater than one conflict per TCB type are candidates for trusted types.

Second, we propose that the analysis also include a security target definition that specifies the *required subject types*. Rather than requiring that the system administrators enumerate all subject types individually, we can use the type transition hierarchy to estimate the set of

Class	*Description*
TCB or Candidate	Trusted subject types
Exclude Type	Type can be excluded from secure system with this TCB
Sanitize	A sanitized read may be used to protect TCB
Denial	Denial of conflicting rights can be used to protect TCB
Modify Policy	Policy must be edited to protect TCB

Table 1: Classifications for TCB integrity conflicts.

types required as all subject types that may transition to a required subject type. Type attributes or other semantically meaningful identifiers can be used to identify desired subject type sets. If a subject type does not belong to the set of required subject types it can be considered for exclusion and the other remaining handling methods.

Identifying object types that may be excluded is more difficult. If we are too ambitious, we may remove an object type that is really needed by the system. In general, when we exclude a subject type, we may remove object types depend on the existence of this subject type. For example, if we remove X windows subject types, we no longer need X windows object types. This may prevent integrity violations for TCB subject types with broad rights, such as the system administrators. The dependency of a subject type on the availability of particular object types is not currently identified. All we know are the operations that can be performed. A conservative approximation is the object types for which objects can only be created by the excluded subject type. Without the subject type, objects of this type would not exist in the system. We have to account for all possible ways of making objects of this object type, including relabeling (specifically *relabelto* permission).

Third, some Biba integrity violations involve reading low integrity that the subject type can actually handle, such as requests for operations. The Clark-Wilson integrity policy accounts for these by allowing transformation processes (TPs) to operate low integrity data (unconstrained data items or UDIs) and even convert them to high integrity data (constrained data items or CDIs). We refer to the ability to correctly function given UDI input as *sanitization* of this input. In Clark-Wilson, TPs must be certified to perform their tasks. We identify both where TPs require sanitization and where they must handle CDIs properly. Our initial assumption is that all data used by TCB subject types are CDIs, but some data may be downgraded to UDIs and used via sanitization.

Recall the distinction between read integrity and read-write integrity violations. We state that read integrity violations may be sanitized, but read-write integrity conflicts have no possibility of sanitization (i.e., data written by a TCB subject type is always a CDI, in Clark-Wilson terms). Recall that read-write integrity violations mean that the subject type writes and reads data that can be modified by a lower integrity subject type. Depending on synchronization, a lower integrity subject type may be able to change an objects as the higher integrity subject type is writing them. While sanitization may be possible in general, we flag these violations as being beyond sanitization.

Fourth, the read-write integrity violations are classified for ad hoc denial of rights. In many cases, more rights are assigned than are really necessary for the application, which is a problem of least privilege. In some cases, it may be sufficient and simpler to simply deny the conflicting rights. Gokyo enables partitioning of conflicts, and assigning independent handling to each partition. Therefore, it is possible to simply denial these rights without further modifying the SELinux example policy. Application-specific examination is necessary to determine if these denials are really possible.

Lastly, if we find that the permission assignment is necessary for the convenient execution of a required application, then modification of the policy is the only remaining option.

3.4 Manual Analysis

Manual analysis involves starting at the highest level handling method and determining whether it can actually be applied. If not, then the subsequent methods must be considered.

Identifying trusted writers and excluded writers can be done automatically, so the main effort here is on determining whether sanitization is possible and identifying the sanitization method. This is a fairly ad hoc process, so we examine it relative to our integrity analysis results in Section 4.

If sanitization is not possible, then expressing a denial for this exception or policy modifications are the remaining options. Both of these must be done manually at present.

4 Integrity Analysis

In this section, we use Gokyo to analyze our proposed TCB to identify the integrity conflicts, classify according to best possible resolution, and choose the likely resolution. The likely resolution is chosen based on manual analysis of the conflict. The key results are the resultant TCB (i.e., does it need to be expanded and how?) and proposed SELinux policy changes needed to achieve this TCB. Detailed discussion of the Gokyo tool itself is provided elsewhere [13].

4.1 Analysis Implementation

The integrity analysis for the proposed TCB in Section 2.3 is performed on the SELinux example policy for Linux 2.4.19. This policy consists of over 50,000 policy statements [4]. In Gokyo, the SELinux example policy comprises over 700 subject types and type attributes, over 40,000 individual permissions, and over 100,000 explicit assignments between subject types and permissions.

The integrity of the SELinux system is represented by two integrity constraints between the set of proposed TCB subject types and the set of all other subject types as shown in Figure 4. To represent this we create two subject types, TCB subject types (high integrity) and non-TCB subject types (low integrity), and aggregate the subject types into their respective group. The permissions assigned to each subject type node are automatically propagated to the aggregate subject types.

Our integrity protection goal is expressed using two constraints: (1) read-integrity constraint and (2) read-write-integrity constraint. Read-integrity constraints are violated if the low integrity subject has write permission (i.e., a permission representing the ability to modify the data in that SELinux class) to an object type and class pair that high integrity subject type has read permission to. Read-write integrity is violated if the high-integrity subject also has write permission to the object type and class pair in addition to read permission.

[4] Statement count is taken from the macroexpanded policy in `policy.conf`.

To implement these constraints, Gokyo assigns the invalid permissions to the high integrity set. For read-integrity, Gokyo creates a permission with all read permissions assigned for each object type and class pair that the low integrity subject can write. Similarly, for read-write-integrity, Gokyo creates a permission with all write permissions assigned for each object type and class pair that the low integrity subject can write. In this case, constraint verification takes an intersection of the invalid permissions and the ones assigned to the TCB subject types (i.e., different types of constraints may have different algorithms).

Note that it may be more efficient to test this constraint in the opposite manner by determining if the low integrity set has write permissions to object type and class pairs that the high integrity subject can read. At this point, we actually create both integrity test sets, but we should determine which is smaller and test only that one instead.

Analyzing integrity protection is basically a task of identifying all integrity conflicts and classifying them into their best legal classification. We have found that the number of conflicts that exist in the entire SELinux policy is too large to be effectively considered together. Fortunately, conflicts are independent. That is, the existence of one conflict has no effect on another. This means that a classification to eliminate one conflict cannot be undone by another conflict. For example, if we find that we can sanitize the use of a particular conflicting permission, the emergence of a conflict later does not impact this sanitization. This is true for all classifications. The one caveat is that we may find that a particular subject requires so many sanitizations that it should be trusted or excluded, but these cases are not excessive and easily handled. Usually, we determine whether a subject should be trusted or excluded before we do the hard work of figuring out a sanitization.

The result is that we can consider the conflicts in small groups, and make classifications based on a subset of the information. Currently, we use Gokyo in a mode in which it identifies a single conflict for each invalid permission (i.e., constraint-generated). Sometimes, attribute assignments result in multiple, unique conflicts, but Gokyo only presents the attribute assignment once. Gokyo generates a log containing all conflicts and the assignment paths between nodes involved in the conflict, including the line numbers in which the assignments are specified. This assists with the manual analysis phase. However, addition metadata, such as the frequency of conflict for a particular invalid permission, would also be useful. The log of a constraint violation is shown below (`class.conf` is SELinux policy file, `kernel.cst`

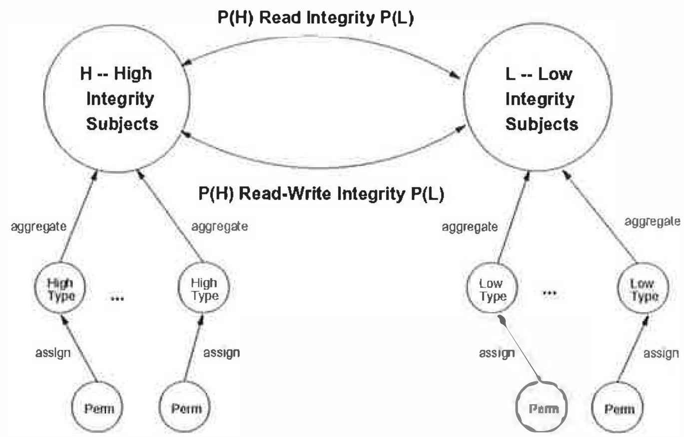


Figure 4: Gokyo graphical policy model implementation of integrity.

Trusted Type	*Conflict Type*	*Object Type & Op*	*Class*	*Resolution*
dpkg_t	tmpreaper_t	tmp_dpkg_t:file rw	exclude	exclude
initrc_t	many	file_type:blk/chr/file r	sanitize	sanitize
initrc_t	useradd_t	etc_t:file r	trust	trust
initrc_t	hwclock_t	clock_device_t:chr/blk rw	trust	trust
initrc_t	gpm_t	psaux_t:chr rw	exclude	exclude
initrc_t	sound_t, xdm_t	sound_device_t:chr rw	trust	exclude
initrc_t	httpd_admin_xserver_t	framebuf_device_t:chr rw	deny	exclude
initrc_t	many	initrc_t:fifo rw	deny	sanitize
kernel_t	slapd_t, squid_t, +	*:*_socket r	sanitize	sanitize
kernel_t	dhcpc_t	resolv_conf_t:file r	trust	exclude
kernel_t	dhcpd_t	var_run_dhcpd_t:file r	trust	exclude
kernel_t	quota_t	file_t:file r	trust	trust
local_login_t	many	proc_t:file r	sanitize	sanitize
local_login_t	insmod_t	local_login_t:process r	exclude	exclude
local_login_t	logrotate_t	local_login_t:process r	trust	trust
mount_t	automount_t	autofs_t:dir rw	exclude	trust
mount_t	bootloader_t, fsadm_t	fixed_disk_device_t:* rw	trust	trust
sysadm_t	user_t	misc_device_t:* rw	deny	exclude obj
sysadm_t	many	sysadm_devpts_t/ptyfile:* rw	deny	change
sysadm_t	sysadm_*_t	sysadm_home_t:* rw	deny	change/sanitize one file
sysadm_t	sysadm_*_t	sysadm_tmp_t:file rw	exclude	change
sysadm_t	sysadm_irc_t	sysadm_irc_t:file rw	exclude	change/sanitize
sysadm_t	sysadm_xserver_t	sysadm_xserver_t:shm rw	exclude	exclude
sysadm_t	sysadm_xauth_t	sysadm_home_xauth_t:file rw	exclude	exclude
sysadm_t	admin	kernel_t:system avc_toggle rw	trust	trust
sshd_t	many	sshd_devpts_t/userpty:* rw	deny	change

Table 2: Integrity conflicts in the initial TCB proposal.

is our constraint file, and `kernel.cfg` contains aggregate subject type definitions):

```
On constraint: kernel.cst(25)
Role 151: mount_t
has constraint: "integrity protected"
with node: Role 882: non-mount

Violating Assignments:
  Permission 2876: autofs_t:dir 00110000
   (1) From: class.conf (60810) Perm 2876:
autofs_t:dir 00110000
   (2) to: class.conf (60759) Role 151: mount_t

Violating Preclusions:
  Permission 45131: autofs_t:dir 003fffff
   (3) From: kernel.cst (25) Role 882: non-mount
   (4) to: class.conf (60759) Role 151: mount_t
   (5) to: class.conf (0) Perm 42608:
autofs_t:dir 003e1c7e
   (6) to: class.conf (60639) Perm 2857:
autofs_t:dir 003e1c7f
   (7) to: kernel.cfg (94) Role 148: automount_t
   (8) to: kernel.cfg (91) Role 882: non-mount
```

The *violating assignments* is the permission assigned to `mount_t` whose integrity may be violated. Line (1) indicates where the permission was assigned to `mount_t`, and line (2) indicates where `mount_t` was identified as a subject. The file `class.conf` is a truncated version of `policy.conf` for SELinux 2.4.19. For the *violating preclusions*, the path for the assignment of the constraint-generated invalid permission is shown. Line (3) refers to declaration of the aggregate subject type (Gokyo-specific), and line (4) is the same as line (2). Line (5) refers to the generated permission (no file line number because it is generated), and line (6) shows the assignment of `autofs_t` permissions to `automount_t`. Lines (7) and (8) show that `automount_t` is assigned to the non-TCB aggregate.

For each conflict, Gokyo estimates the classifications based on: (1) the number of subject type conflicts (for trust); (2) whether the subject type or object type is required, see below (for excluding subject types); and (3) whether the conflict is read-integrity or read-write-integrity (for considering sanitization). Our proposal for removing object types based on whether the object type is created by only excluded subject types has not been implemented yet, so we use the object types required by our focal subject type.

For required subject types, we assumed that the purpose of our system was to run an Apache web server. Thus, we include all Apache subject types (i.e., those starting with `httpd`) and all those subject types that transition to an Apache subject type, directly or indirectly. In addition to our TCB subject types, we require `dpkg_t` (i.e., the Debian package manager), `rlogind`, several user subject types. Ultimately, we will choose to exclude `rlogind`, but include `user_t` in the analysis. Users may be actively involved in script generation (e.g., for personal pages in a corporate server). Because so few other required subject types are found this way, we will add others later. Note that the set of required object types includes the types accessible to the Apache subject types only.

4.2 Analysis Process

Table 2 shows the integrity conflicts that our proposed TCB has with the remaining system subject types and the possible resolutions of these conflicts. The *trusted type* field shows a trusted type that reads input written by an untrusted type. The *conflict type* field shows one or more of the untrusted types in the conflict. The *object type & op* field shows the conflicting data and the rights of the TCB subject type (i.e., read or read-write integrity conflict). The *class* field shows the classification of the conflict. The *resolution* field shows the manual resolution to the conflict.

The integrity conflicts are collected into groups based on the trusted type. First, `dpkg_t` (debian package management) has a common read-write integrity conflict also because `tmpreaper` (cleans temporary file directories) is given broad file access for cleaning up temporary files. `tmpreaper_t` is responsible for few violations, so the classification is *exclude*. This specification is consistent with `tmpreaper`'s task, so the only two alternatives are to trust or exclude `tmpreaper_t`. We manually choose the latter.

Second, `initrc_t` is involved in a read integrity conflict that affects most trusted types: it is given read access to all file data in the system. Since it can read all files, it certainly has an integrity conflict with the lower integrity subjects. However, the read access is to `getattr` for `stat`, so this can be sanitized.

Third, `initrc_t` has several other conflicts. The next two are identified as required and seem necessary, so we add `useradd_t` and `hwclock_t` are added to the TCB. The next three are not really necessary (`gpm_t` for mouse, `sound_t`, and `xdm_t`), so we choose to exclude them. The X window server introduces a number of other integrity issues, so much more work is necessary to have an X windows system running on an integrity-protected TCB. Thus, `httpd_admin_xserver_t` is excluded. Lastly, we determine that read-write integrity access to `initrc_t`'s fifo can be sanitized as necessary. It should involve only simple communication (e.g., on process start). Note that this is a manual override of our intended requirements.

Fourth, kernel_t has several integrity conflicts with receiving network data. This integrity conflict is common to most services in the TCB. Such interaction is necessary for convenient execution, so we will examine sanitization of network communication in Section 4.3. The other conflicts are so common that the framework assumes that they are trusted. Manual analysis keeps on quota_t (file quota management) in the TCB and excludes dhcpc_t and dhcpd_t.

Fifth, the conflict over access to /proc is found for local_login_t. Since this access is for reading only, we will aim to sanitize this access. Next, we assume that installing modules is not necessary for our secure system, so insmod may be excluded. On the other hand, logging is an important process, so logrotate is added to the TCB.

Sixth, mount_t has conflicts with automount_t, fsadm_t, and bootloader_t. Although only the latter two are common conflicts, all of these are added to the TCB.

Seventh, there are a variety of conflicts with sysadm_t. sysadm_t has a conflict over access to misc_device_t with user subjects. These object types will be excluded. Also, access to sysadm_devpts_t is shared with many subject types. Many of these subjects are application-specific administrators which are intended to be of lower integrity. A different object type should be designated for these. Next, sysadm_t has read-write integrity conflicts with the application-specific administrators over the sysadm_home_t. Conflicting access is provided to permit lower-integrity administrative processes to write to an error log (.xsession-errors). We recommend breaking the object type into two for the higher and lower integrity home objects, so access to the latter can be sanitized. Since we have excluded X windows this object type can also be excluded in this case. Access to sysadm_tmp_t and sysadm_irc_t should be changed similarly. Finally, sysadm_t has conflicts that can be excluded for X windows subject types and trusted for administrative subject types. The following subject types are added to the TCB: ipsec_mgmt_t, apt_t, and admin_passwd_exec_t. install_menu_t is excluded.

Lastly, sshd_t has a read-write integrity conflict over the use of pseudo-terminals. Type change is used for some to change the subject type to a lower-integrity subject upon use of a user pty for sysadm_t, so we presume that this should be added for sshd_t as well.

After the trusted types, excluded types (including object types), and sanitized accesses are added to their respective lists, the next iteration of the analysis can be performed. After some number of iterations, 5 in our case, all the exclusions, sanitizations, and trusted subject types are accounted for, and no conflict remains unclassified. However, resolving the efficacy of sanitizations and reduce file read permissions (or at least managing them) remain.

4.3 Analysis Findings

The base TCB for the SELinux example policy for supporting an Apache is shown in Table 3. Note that the set of subject types that Apache must ultimately depend on would be somewhat larger (around 50% larger given our analysis [13]). Starting with our original 12 types, we have found that 30 subject types must be trusted. The correctness of this TCB depends on the resolution of the full access that these subject have to application and user files which they should probably rarely access, as discussed below. Also, not all forms of authentication are necessary at once. Ultimately, it is probably possible to reduce this set slightly, but this provides a good estimate of most SELinux TCBs.

Interestingly, not long after this paper was submitted, Wayne Salamon independently proposed a "core policy" to the SELinux community [19]. The intent of this proposal was to define a base system policy upon which any other system policies would be derived. There is some notion of usability here rather than TCB, as the intent is for base function rather than base security. After some discussion with the community he settled on 40 policy files (roughly equivalent to 40 subject types) to comprise a core policy. 17 of the subject types in the TCB are common to the two groups. The ones that we include that are not in the core policy proposal are indicated in Table 3. We think that many of the subject types in our proposal are TCB subject types, although some authentication subject types, such as sshd_t, and administrative types, such as sysadm_t and dpkg, are not necessarily core.

As part of the analysis, we identified subject types and object types for exclusion from our system. The 25 subject types we excluded are listed in Table 4. We need to verify empirically that such services are not actually necessary for an Apache system on SELinux, but most of these do not seem controversial.

In Table 5, we summarize the sanitizations required for our TCB. Note that in Clark-Wilson terms, these saniti-

kernel_t*	init_t	initrc_t	sysadm_t*	getty_t
mount_t	fsadm_t	load_policy_t	dpkg_t*	checkpolicy_t
setfiles_t	syslogd_t	klogd_t	automount_t	sshd_t*
sshd_login_t*	local_login_t	quota_t*	ldconfig_t	useradd_t
hwclock_t*	apt_t*	cardmgr_t*	ipsec_mgmt_t*	admin_passwd_exec_t*
bootloader_t	logrotate_t	newrole_t	snmpd_t*	passwd_t*

Table 3: Final trusted computing base subject types (* indicates not in proposed SELinux core policy).

insmod_t	rlogind_t	remote_login_t	sysadm_xserver_t	xdm_t
sysadm_xauth_t	sound_t	tmpreaper_t	httpd_admin_xserver_t	kmod_t
lpd_t	xdm_xserver_t	vmware_user_t	sendmail_t	procmail_t
hotplug_t	traceroute_t	update_modules_t	gatekeeper_t	smbd_t
dhcpc_t	dhcpd_t	install_menu_t	devfsd_t	gpm_t

Table 4: Final excluded subject types.

zations indicate the unconstrained data items (UDIs) that our TCB's transformation procedures (TPs) must handle. By sanitization, we envision that SELinux modules can be annotated with sanitization policies to verify the format of the inputs. This is a non-trivial endeavor, so such a proposal is quite preliminary. However, such sanitization services on top of a verified and simple integrity policy can enable fulfilling of our security goals without major policy tweaking.

Some of these sanitizations are focused and can be handled as exceptions, but some (the first four) have many instances. Our impression is that the fifos can be handled because each instance serves the same purpose. Socket access is both extensive in number of communicators and variety of communications. Significant effort is required to comprehensively address these. Most of the information in `/var` and `/proc` does not seem to impact the processing of our trusted subjects, but more investigation is necessary.

The two conflicts that remain are: (1) between trusted subject types and the pseudo-terminals that they share with user process and (2) the permission assignments that permit trusted subjects to access to all files (the first and last entries in Table 2). The first conflict is probably best handled by a SELinux *type change* statement. These are used to change the type of an object based on the subject type of the accessor. When a pseudo-terminal is accessed by a high integrity subject, it gets a high integrity type and its previous state is cleared.

The second conflict could be addressed by leveraging Gokyo. Using Gokyo's conflict spaces, we could declare auditing or intrusion detection to be initiated when an integrity-conflicted file object is accessed by a high-integrity subject. This would not require a modification to the SELinux policy, but a Gokyo conflict space assignment would be necessary [13]. Such a solution depends on infrequent use of conflicting permissions. If there is frequent use of some conflicting permissions, then alternative measures are needed. This task remains as future work.

Note that a SELinux *auditallow* statement would not quite work in this case because it would audit all file accesses instead of the ones that violate integrity. Of course, we could always modify the SELinux policy, but this would take significant effort and perhaps lead to further conflicts.

5 Related Work

SELinux includes tools for policy analysis. *neverallow* statements enable the policy designer to express assignments that should not be expressed in the policy. The *checkpolicy* tool verifies that no *neverallow* statement is violated when the policy is compiled. Such statement enable the identification of conflicts, but any resolution requires changing the SELinux policy directly. These statements are suitable for expressing cases that should not ever occur, but they are not suitable for expressing high level security goals.

The Tresys Corporation has been developing tools for analyzing SELinux policies for over one year [23]. Tresys defines tools for helping administrators understand the SELinux policy (e.g., statements for a par-

Object Type	*Sanitization*
*_t:fifo	Mainly for use following exec
*:*_socket	Must be able to handle network data or big policy mod
proc_t:file	Mainly expected to print this information
sysadm_home_t:*	Only need to read .xsession-errors log

Table 5: Sanitized conflicts and brief analysis.

ticular subject type) and helping perform administrative tasks (e.g., correctly adding a new user). Such tools are valuable for the development of SELinux policies, but do not address the questions of whether the policies can meet particular high-level goals.

We are aware of work ongoing at MITRE to analyze SELinux policies for more complex relationships, such as reachability [7]. The SELinux example policy is so large that the theorem proving tools being used are not efficient enough for effective analysis yet.

Access control policy analysis itself is a fairly recent area of work. Bertino et al define a general framework for representing and reasoning about access control models [3]. The goal here is to compare models (e.g., for expressive power) rather than compare policies to properties. We believe that their model is expressive enough to do the latter, however.

Further, Jajodia et. al. [14] support conflict resolution in their model. In their case, the goal is to find a general strategy of conflict resolution, not to support different strategies. Ferrari et. al. [8] examine conflict resolution problems and strategies as well.

6 Conclusions

In this paper, we present an approach for analyzing integrity protection of the SELinux example policy. The SELinux module supports the recent Linux Security Modules (LSM) framework for implementing mandatory access control on the Linux kernel. The SELinux example policy is undergoing active development and is being applied in several installations. The aim is for administrators to take the SELinux example policy and customize it to their site's security goals. This quite difficult, however, because the SELinux policy model is quite complex and the SELinux example policy is large.

Our aim is to provide an access control model to express site security goals and resolve them against the SELinux policy. In particular, we want to identify a minimal system TCB for the SELinux example policy that satisfies Clark-Wilson integrity restrictions relative to the rest of the system. UNIX systems are not designed to meet Biba integrity, but the Clark-Wilson integrity policy enables a description where key data can be identified (those data used by TCB subject types), and sanitization of low integrity data is possible.

We have developed a tool called Gokyo that represents the SELinux example policy and our integrity goals, identifies conflicts between them, estimates the resolutions to these conflicts, and provides information for deciding upon a resolution. Further, Gokyo represents the state of the integrity resolution which could be used by the access control module to make authorization, audit, and intrusion detection decisions. Using Gokyo, we found a minimal TCB containing 30 subject types that meets Clark-Wilson integrity including sanitization requirements and resolution of overly broad file access rights. More investigation is needed to verify the proposed sanitization requirements and determine the effectiveness of audit versus restriction of file rights, but the Gokyo's ability to support the analysis of integrity protection is helpful in understanding and managing higher level security goals on complex policies.

Acknowledgements

The authors would like to thank the anonymous referees for their useful comments, and those people participating in the SELinux community, particularly Stephen Smalley and Russell Coker.

References

[1] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat. A Domain and Type Enforcement UNIX Prototype. In *Proceedings of*

the 1995 USENIX Security Symposium, 1995. Also available from TIS online archives.

[2] D. Bell and L. La Padula. Secure Computer Systems: Mathematical Foundations (Volume 1). Technical Report ESD-TR-73-278, Mitre Corporation, 1973.

[3] E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A logical framework for reasoning about access control models. *ACM Transactions on Information and System Security (TISSEC)*, 5(4), Nov 2002.

[4] K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, Mitre Corporation, Mitre Corp, Bedford MA, June 1975.

[5] W. E. Boebert and R. Y. Kain. A Practical Alternative to Hierarchical Integrity Policies. In *Proceedings of the 8^{th} National Computer Security Conference*, Gaithersburg, Maryland, 1985.

[6] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. Proceedings of the 1987 IEEE Symposium on Security and Privacy, 1987.

[7] A. Herzog. Personal communication.. November 2002.

[8] E. Ferrari and B. Thuraisingham. Secure database systems. In O. Diaz and M. Piattini, editors, *Advanced Databases: Technology and Design*, 2000.

[9] T. Fraser. LOMAC: Low Water-Mark Integrity Protection for COTS Environments. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, May 2000.

[10] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8), August 1976.

[11] T. Jaeger and J. E. Tidswell. Practical safety in flexible access control models. *ACM Transactions on Information and System Security (TISSEC)*, 4(2), May 2001.

[12] T. Jaeger, A. Edwards, and X. Zhang. Managing access control policies using access control spaces. In *Proceedings of the 7^{th} ACM Symposium on Access Control Models and Technologies*, June 2002.

[13] T. Jaeger, A. Edwards, and X. Zhang. Policy management using access control spaces. *ACM Transactions on Information and System Security (TISSEC)*, to appear.

[14] S. Jajodia, P. Samarati and V. Subrahmanian. A Logical Language for Expressing Authorizations. *Proceedings of the IEEE Symposium on Security and Privacy*, 1997.

[15] P. Karger and R. Schell. Thirty years later: Lessons from the Multics security evaluation. IBM Technical Report, RC 22534, Revision 2, September 2002.

[16] P. Loscocco, S. Smalley, P. Muckelbauer, R. Taylor, J. Turner, and J. Farrell. The inevitability of failure: The flawed assumption of computer security in modern computing environments. *Proceedings of the 21^{st} National Information Systems Security Conference*, October 1998.

[17] S. Minear. Providing policy control over objects in a Mach-based system. *Proceedings of the Fifth USENIX Security Symposium*, 1995.

[18] National Security Agency. Security-Enhanced Linux (SELinux). http://www.nsa.gov/selinux, 2001.

[19] W. Salamon. Core policy, second pass. SELinux mailing list archives, http://www.nsa.gov/selinux/list-archive/3941.html, 2003.

[20] S. Smalley. Configuring the SELinux policy. NAI Labs Report #02-007, available at www.nsa.gov/selinux, June 2002.

[21] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, and J. Lapreau. The Flask security architecture: System support for diverse policies. *Proceedings of the Eighth USENIX Security Symposium*, August 1999.

[22] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux Security Modules: General security support for the Linux kernel. *Proceedings of the Eleventh USENIX Security Symposium*, August 2002.

[23] Tresys Technology. Security-Enhanced Linux research. www.tresys.com/selinux.html, 2001.

Security holes... Who cares?

Eric Rescorla
RTFM, Inc.
ekr@rtfm.com

Abstract

We report on an observational study of user response following the OpenSSL remote buffer overflows of July 2002 and the worm that exploited it in September 2002. Immediately after the publication of the bug and its subsequent fix we identified a set of vulnerable servers. In the weeks that followed we regularly probed each server to determine whether its administrator had applied one of the relevant fixes. We report two primary results. First, we find that administrators are generally very slow to apply the fixes. Two weeks after the bug announcement, more than two thirds of the servers were still vulnerable. Second, we identify several weak predictors of user response and find that the pattern differs in the period following the release of the bug and that following the release of the worm.

1 Introduction

It's widely believed that users are not particularly diligent about installing security fixes to their software. For instance, Netcraft [1] reports that 75% of users of SSL-ized versions of Apache had failed to upgrade their servers months after the Apache chunked encoding vulnerability [2] was announced. [1] Despite this belief, an enormous amount of effort has been put into the discovery, repair, and disclosure of security flaws in software. The implied rationale for all this work is that users who are informed about the security flaws of their software can upgrade as appropriate.

In this paper, we describe the results of direct measurement of deployment of fixes for the popular Open Source SSL/TLS toolkit OpenSSL, as installed in the mod_SSL package for the Apache Web server. For a number of reasons we would expect mod_SSL users to be better than average about installing security fixes:

- OpenSSL is security software and therefore its users clearly desire security.
- OpenSSL users are overwhelmingly UNIX users and UNIX users are widely believed to be more experienced in server administration than Windows users.
- Many popular operating systems (Linux, *BSD) have packages to make installing OpenSSL easier.
- We are studying the deployment of OpenSSL in servers which are particularly vulnerable because they must be open to the Internet at all times.
- The flaw allowed an attacker to take over the entire Web server.

In spite of all these factors, our measurements show remarkably slow deployment. One week after the flaw was announced, only 23% of the servers under study had been fixed. Two weeks after, less than 1/3 had been fixed. At the time of release of the Slapper worm that exploited this vulnerability [3], almost 60% of servers were still vulnerable. Figure 1 shows the percentage of vulnerable servers for the period under study.

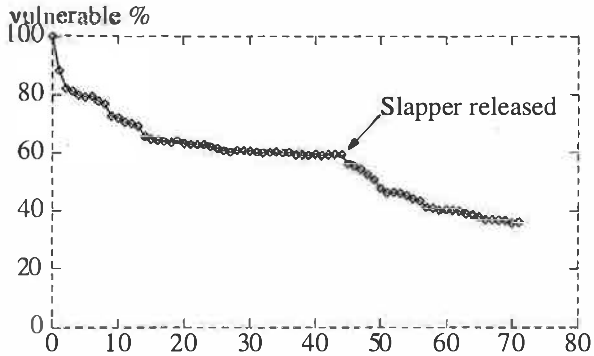


Figure 1 Vulnerable servers over time

2 Disclosure of the Vulnerabilities

On the morning of July 30, 2002, Ben Laurie, a member of the OpenSSL core team, sent an advisory entitled "OpenSSL Security Alert - Remote Buffer Overflows"[4] to a number of Internet mailing lists, including openssl-users and bugtraq. This announcement described the following flaws in OpenSSL:

1. The SSLv2 CLIENT-MASTER-KEY message was being improperly processed by servers. An overly-long message could be used to overrun a buffer on the heap. This bug was known to be exploitable.
2. An overlong SSLv3 SessionID value supplied by the server could be used to overrun a buffer on the client.
3. An overlong SSLv3 master key supplied to a server could cause overflow. This bug applied only to beta versions of OpenSSL 0.9.7.
4. Various buffers for ASCII representations of integers were too small on 64 bit platforms.
5. Integer overflows in the ASN.1 parser could be exploited by supplying it with certain invalid encodings. This could cause a denial of service attack.

The most important of these bugs were (1) and (2). Bug (1) would allow compromise of any OpenSSL server running SSLv2 (nearly all do) and bug (2) would allow compromise of any OpenSSL client running SSLv3 (nearly all do). Bug (1) was particularly serious because any attacker could connect directly to a vulnerable server and compromise it, whereas bug (2) could only be exploited if the client would be induced to connect to the attacker's server. This is more difficult but by no means impossible. Since bug (1) was more serious and easier to measure we will be discussing it for the rest of the paper.

2.1 Dissemination

The wide popularity of OpenSSL led to extensive publicity almost immediately. The bug was widely reported in the press and within a day or two, operating system and application vendors began announcing versions of the advisory customized for their platform, along with instructions for upgrading. Figure 2 shows a timeline of major announcements.

July 30, 2002 10:08 GMT	Initial announcement
July 30, 2002 10:15 GMT	Patches for other versions released
July 30, 2002 11:08 GMT	OpenSSL 0.9.6e available for FTP
July 30, 2002 12:47 GMT	Debian announces [5]
July 30, 2002 13:00-12:00 GMT	Trustix, Engarde, Gentoo [6, 7, 8]
July 30, 2002 13:59 GMT	Vulnerability posted to Slashdot [9]
July 31, 2002 17:43 GMT	FreeBSD announces [10]
August 2, 2002 15:17 GMT	Apple announces [11]
August 2, 2002 17:33 GMT	NetBSD announces[12]
August 9, 2002 11:54 GMT	OpenSSL 0.9.6g released

Figure 2 Timeline of disclosure events

2.2 Scope of the Bug

At the time of the advisory there was a lot of confusion about which software was affected. Essentially, this confusion came in two forms:

1. Confusion about which SSL-using software was affected.
2. Confusion about which OpenSSL-using software was affected.

The first form of confusion stemmed from the fact that the flaws could be exploited by sending bogus SSL protocol messages and therefore the flaws were associated with SSL. No doubt the confusion was amplified by the prior discovery of problems with Internet Explorer's Basic Constraints processing [13] which was also associated in people's minds with SSL and all implementations thereof.

Although OpenSSL is the dominant free SSL implementation, many widely used SSL-using packages do not use OpenSSL. In particular, Internet Explorer, Internet Information Server, and Mozilla do not use OpenSSL and are therefore not vulnerable to this attack. (Despite being Open Source, Mozilla uses Netscape's home grown Netscape Security Services library rather than OpenSSL).

The second form of confusion resulted from the widespread use of OpenSSL as the cryptographic library for a wide variety of other applications. Since the flaws were primarily in the SSL protocol implementation, these applications were not affected. However, this fact was not widely understood and many vendors recommended upgrading not only OpenSSL but any application which depended on it. For instance, the OpenPKG advisory [14] recommended reinstalling OpenSSH, scanssh, and tcpdump, which used OpenSSL's cryptographic functions but not its SSL implementation or ASN.1 routines. It seems likely that some users who would otherwise have upgraded did not because they believed it to be a much larger job than it in fact was.

2.3 Round Two

The security fixes made to OpenSSL included a number of assertions designed to check for various illegal protocol conditions. These conditions were not known to be exploitable but checking for them was clearly an improvement. Unfortunately, the assertions were structured explicitly as C `assert()`s and failure caused the program to exit. This made it rather easy to mount a DoS attack on an SSL server by feeding it invalid data and thus causing a crash.

In order to fix this bug, the OpenSSL team released OpenSSL 0.9.6f on August 8, which replaced the assertions with errors. Due to inadequate preparation time, there were a number of build problems with 0.9.6f and the team was forced to issue 0.9.6g to solve those problems. The OpenSSL team did not issue patches for these problems but a number of operating system vendors did. As of the end of the study period, 0.9.6g was the latest stable version.

2.4 The Slapper Worm

On September 13th at 13:55 GMT, Fernando Nunes announced [3] that a worm had compromised his machine via the SSLv2 hole described here. The existence of this worm was independently verified and it was soon dubbed Slapper. The details [15] of the Slapper worm aren't important to understand, so we will simply summarize the important points.

Vulnerable Systems

Slapper randomly chooses IP addresses and then probes them in an attempt to determine the server version. When it finds a vulnerable server, it uses the SSLv2 buffer overflow

to install itself on the target. Since each version of Apache requires a slightly different exploit, Slapper contains a table of version/memory offset pairs. If the version is not in the table, Slapper attempts to attack it anyway, using a default version guess, but this likely just creates a crash rather than a compromise, unless the guess happens to be correct. As of this writing, Slapper only attacks Linux machines, but it could be easily adapted for other platforms.

Worm Behavior

Once Slapper has compromised a target machine, it then does two things. First, it joins a peer-to-peer network of other worm instances. This allows the worm to be remotely controlled. The worm can then be used to DoS attack other systems or damage the compromised machine. In addition, the worm then looks for other machines to infect.

3 Countermeasures

As usual, the advisories admonished users to apply fixes as soon as possible. Users had four choices:

1. Turn off OpenSSL entirely.
2. Disable SSLv2 (server-only).
3. Upgrade to a newer fixed version.
4. Install one of the patches (supplied with the advisory), which fixed the vulnerabilities but did not upgrade to the newer version.

Turning off SSL support isn't a realistic option for most sites but it's worth considering the other three options.

3.1 Disabling SSLv2

Turning off SSLv2 is by far the simplest possible countermeasure, and is effective on all 32-bit platforms. In both mod_SSL and ApacheSSL, a simple configuration directive will turn off all support for SSLv2. All the administrator needs to do is edit the configuration file and restart the server. Although administrators are sometimes concerned about being unable to serve SSLv2-only clients, SSLv3-capable clients have been available since 1996 and SSLv2-only clients are extremely rare. Therefore, the negative side effects of this countermeasure are quite minimal.[2]

3.2 Upgrading

The OpenSSL project currently maintains two source branches, the stable 0.9.6 branch and the beta-level 0.9.7 branch. Concurrent with the security alert, the OpenSSL team released versions 0.9.6e and 0.9.7-beta3. In both cases the releases were both source and binary-compatible with the previous releases in that branch, though not necessarily with other branches.

OpenSSL is extremely portable and so a source-level upgrade is quite easy. Assuming that the new version is compatible with the current version, the user merely needs to do:

```
./config
make
make install
```

In many cases, applications are dynamically linked against OpenSSL and so simply reinstalling OpenSSL is sufficient. If applications are statically linked they must be recompiled.

3.3 Patches

The OpenSSL project provided patches for all major versions of OpenSSL. We have test-applied them and they seem to work fine. In many cases, operating system vendors also supplied patches for the versions they had shipped with their system.

3.4 Operating System Vendor Behavior

One of the primary channels for deployment of OpenSSL is via operating system vendors. NetBSD, FreeBSD, and OpenBSD all provide OpenSSL as part of their base system. All the major Linux vendors provide packages (RPMs or .debs) for some version of OpenSSL. As a consequence, many users have never had the experience of building OpenSSL from source and expect to get their updates from their Operating System vendor. Figure 3 shows the updates provided by the major Open Source vendors.

Vendor	Update Type	Binary
Debian	Patch (deb)	Yes
Engarde	Patch (RPM)	Yes
OpenPKG	Patch (RPM)	No
SuSE	Patch (RPM)	Yes
Red Hat	Patch (RPM)	Yes
Mandrake	Patch (RPM)	Yes
Connectiva	Patch (deb)	Yes
Caldera	Upgrade (RPM)	Yes
OpenBSD Patch	CVS	No
NetBSD	Both (CVS)	No
FreeBSD	Patch (CVS)	No

Figure 3 Updates provided by various vendors

As noted above, all the major Open Source vendors were extremely proactive about delivering updates. In every case, updates were available within 2 days of the initial advisory.

As should be apparent from Figure 3, updating on Linux was rather easier than updating on *BSD, since all of the *BSD updates required compilation, either of the base system or from the ports/packages collection.

4 Methodology

Our method is conceptually simple. We first assembled a list of SSL-capable servers and then periodically probed each server to determine whether or not it had deployed the fix, as well as what kind of fix (disabling SSLv2, patch, or upgrade) had been deployed.

4.1 Assembling the Corpus

It was first necessary to obtain a large list of SSL-capable servers. Although a number of different types of servers run SSL, we restricted ourselves to HTTPS (Web) servers because they are by far the most common and the easiest to find. In addition, using only HTTPS servers allows us to have a more homogeneous sample and thus to draw tighter conclusions, at least for this limited domain.

Following the method of Murray [16] we built our list using a search engine. We randomly selected a list of 10,000 words from FreeBSD's `/usr/share/dict/words` file and then queried Google for each word using a search for "https and <word>" with the limit of hits per page set to 100. We processed each page to find all the URLs on the page beginning with "https://" and built a list of all hosts which were potential SSL servers. We contacted each server on this list and selected those which responded to an SSL query (with a 5 minute timeout) and which advertised that they used OpenSSL (thus limiting the sample size to mod_SSL and its derivatives since ApacheSSL does not advertise OpenSSL version). To ensure that we use the same server every time we sample by IP address, not DNS name. Therefore, we collapsed all servers with the same IP address into a single entry. This left us with a sample size of 893 servers. During the study, two administrators complained[3] to us about being probed and we removed them from the sample group, for a final sample size of 891.

4.2 Probing

Our task is to determine which of actions any given administrator has taken. To do this, we need three pieces of information:

1. What OpenSSL version is the server running?
2. Does the server support SSLv2
3. Is the server vulnerable?

Given these pieces of information the decision procedure, shown in Figure 4, is relatively straightforward.

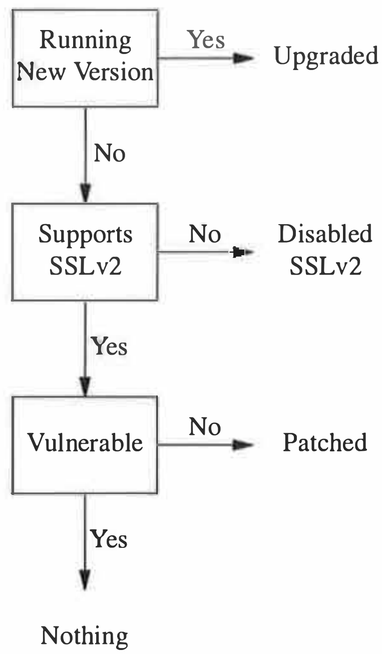


Figure 4 Determining what the administrator has done

OpenSSL Version

First we need to determine the version of OpenSSL that is running. mod_SSL and its derivatives advertise the OpenSSL version that they were compiled with in the HTTP `Server:` header line. We therefore use this as a proxy for OpenSSL version. Thus, we can simply connect to the HTTPS server and issue a `HEAD` request. The server responds with an HTTP header containing the `Server:` field and hence the answer we desire:

```
Server: Apache/1.3.26 (Unix) mod_ssl/2.8.10 OpenSSL/0.9.6
```

Note that if OpenSSL has been upgraded without recompiling the server, this version will not be correct. See Section 7 for more discussion of this topic.

SSLv2 Support

Determining whether the server supports SSLv2 is equally easy. We simply configure our client to offer only SSLv2 and then attempt to handshake with the server. A success means that the server supports SSLv2. A failure means it does not.

Detecting Vulnerable Servers

The patches provided by the OpenSSL project do not change the version number. As a consequence, we cannot simply examine the advertised version number in order to determine whether or not they have been applied. However, due to the nature of the SSLv2 problem, it's still relatively straightforward to determine whether or not a server is vulnerable and therefore by excluding servers which advertise a newer version number identify whether or not a given server has been patched.[4]

The message used to exploit the OpenSSL SSLv2 problem is the CLIENT-MASTER-KEY message, which has the syntax:

```
char MSG-CLIENT-MASTER-KEY
char CIPHER-KIND[3]
char CLEAR-KEY-LENGTH-MSB
char CLEAR-KEY-LENGTH-LSB
char ENCRYPTED-KEY-LENGTH-MSB
char ENCRYPTED-KEY-LENGTH-LSB
char KEY-ARG-LENGTH-MSB
char KEY-ARG-LENGTH-LSB
char CLEAR-KEY-DATA[MSB<<8|LSB]
char ENCRYPTED-KEY-DATA[MSB<<8|LSB]
char KEY-ARG-DATA[MSB<<8|LSB]
```

The first two fields are fixed length—simply a message type followed by the cipher suite that the client has chosen. The rest of the message consists of the *clear key* (used in export mode), the *encrypted key*, which is simply a random value encrypted in the server's public key, and the *key arg*, which is used as the block cipher IV. In the CLIENT-MASTER-KEY message, the length of each field is specified first, followed by the concatenated values of all three fields. Thus, the last bytes of the message are the value of the key-arg field.

The source of the vulnerability is that the key arg is a fixed-length buffer in the OpenSSL `session` structure. All of the block ciphers specified in SSLv2 have 64-bit block sizes and therefore the maximum size of the key arg is 8 bytes. When the CLIENT-MASTER-KEY message is read in by the function `get_client_master_key()`, this length is never checked and therefore an over-long key arg can overrun the buffer and cause a heap overflow. Since OpenSSL uses function pointers extensively, a heap overflow is easy to exploit. The fix for the problem is simply to check the length value of the key arg prior to copying it into the buffer, as shown in the code in Figure 5.

It's possible to determine whether OpenSSL has been patched by generating a CLIENT-MASTER-KEY which is one octet too long. It's easy to see that in a patched version, this falls afoul of the length check shown in Figure 5. The result is a handshake error.

```
n2s(p,i); s->session->key_arg_length=i;
if(s->session->key_arg_length > SSL_MAX_KEY_ARG_LENGTH)
        {
        SSLerr(SSL_F_GET_CLIENT_MASTER_KEY,
               SSL_R_KEY_ARG_TOO_LONG);
        return -1;
        }
```

Figure 5 Length check in OpenSSL 0.9.6e

In an unpatched version of OpenSSL, the server overruns the `key_arg` with whatever the extra byte is. However, as we shall now show, this overrun is harmless.

Once the overrun has occurred there are two pieces of incorrect data:

1. The next field in the `session` structure is now damaged.
2. The `key_arg_length` is 9 instead of 8

Fortuitously, both of these changes are harmless. Assuming normal structure ordering the next field in the `session` structure is the `master_key_length` (with sparse field layout the overrun may damage empty data). This field is set at the end of `get_client_master_key()` and is not used elsewhere in the function and so the damage goes unnoticed.

The `key_arg_length` field is never used elsewhere in OpenSSL. As each encryption algorithm knows its own block size, the `key_arg` buffer is passed directly into the encryption routines as an IV without a length attached. As a result, all the keying material remains the same and the handshake completes successfully. It's therefore easy to detect an unpatched OpenSSL server by simply using an overlong key arg and checking to see if the handshake completes.

4.3 Differentiating Fixed Versions

Although the OpenSSL team did not distribute patches for the DoS vulnerabilities, a number of vendors appear to have done so. If the fixes introduced with 0.9.6g had been restricted to fixing the DoS vulnerability, it would not have been possible to probe for these updates without damaging the servers in question. However, the updates also slightly modified the behavior of the server to the vulnerability probe described in Section 4.2. Figure 6 shows the overlong key-arg test found in 0.9.6g.

```
n2s(p,i); s->session->key_arg_length=i;
if(s->session->key_arg_length > SSL_MAX_KEY_ARG_LENGTH)
        {
        ssl2_return_error(s,SSL2_PE_UNDEFINED_ERROR);
        SSLerr(SSL_F_GET_CLIENT_MASTER_KEY, SSL_R_KEY_ARG_TOO_LONG);
        return -1;
        }
```

Figure 6 New key arg length check

Note the call to `ssl2_return_error()`. This generates an SSLv2 error message on the wire, whereas the previous check merely closed the connection. It is therefore possible to detect the newest round of fixes by the presence of this message.

4.4 Sampling Procedure

The primary work of sampling is done by two programs, `id-client` and `hstest`. `id-client` attempts the `HEAD` request described in Section 4.2 and `hstest` simply attempts a handshake and records whether it succeeds or fails. For historical reasons, both version detection and SSLv2 testing are done with `id-client` and patch detection is done with `hstest` . SSLv2 testing could be done just as well with `hstest`.

We sampled four times daily at 6 hour intervals. We tested version advertisement and SSLv2 support four times a day, at 6:00 AM, noon, 6:00 PM, and midnight Pacific. Because the patch detection creates a noticeable signature on the target server, which often annoys administrators, we test only twice a day, at 6:00 AM and 6:00 PM. In a number of cases we've encountered network or other problems during testing, in which case we've collected data as soon as possible thereafter. The hourly variance in performance is small enough that we believe this does not seriously affect the data.

5 Results

This section describes the data collected during the four weeks following the original announcement.

5.1 Baseline

In our first complete sample, taken at July 30, 16:43 GMT, our sample showed evidence of the deployment of 14 separate versions of OpenSSL. The vast majority of these versions were from the stable 0.9.6 branch. Figure 7 shows the distribution.

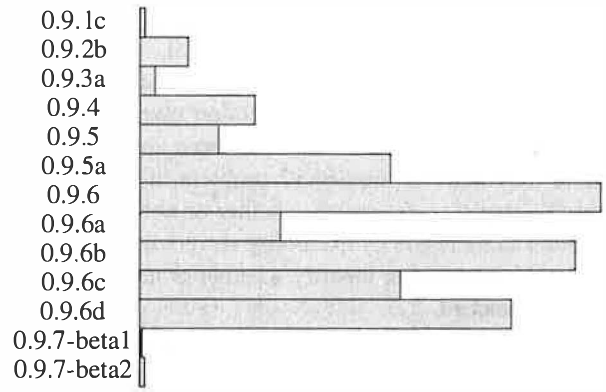


Figure 7 Version distribution at bug discovery

Note that version 0.9.6d was not the most popular version at bug discovery time.

5.2 Propagation of Fixes

The first question we want to ask is: how fast do administrators act to close security holes? Since we directly measure the vulnerability of a given server, this question is easy to answer, as shown in Figure 8. Figure 8 shows the percentage of servers in our sample each day that were vulnerable. Note that we have trimmed the y axis to start at 60% to focus on the on the relevant sections of the data.

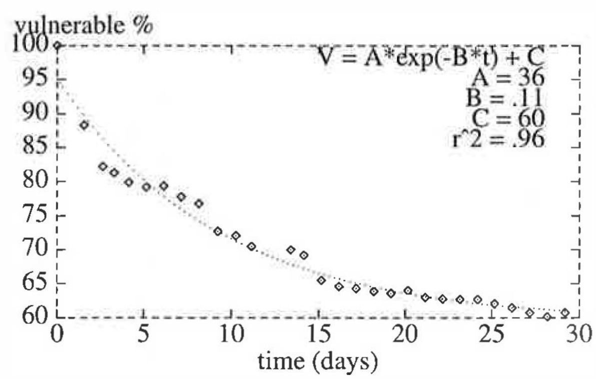


Figure 8 Vulnerable servers as a function of time

The obvious model for this data is an exponential decay curve. We have fitted this data to $vulnerable = Ae^{-Bt} + C$, producing the curve shown in Figure 8. Installation of security fixes is heavily frontloaded. A very large number of people upgraded immediately, with 16% of the sample population installing fixes within the first day and 20% within the first week. However, after that time, deployment rapidly falls off with the deployment curve going almost totally flat after three weeks.

As shown in Figure 8, roughly, 2/3 of the users who were going to install fixes had done so within two weeks. This suggests that the population is strongly bimodal, with roughly 35% of the population closely monitoring security updates and upgrading immediately and the remaining 65% taking no action of any kind.

Note that Figure 8 shows three big jumps, on days 0-1, days 6-7, and days 12-13. The first jump is a genuine mass upgrade event but the second two jumps are artifacts, reflecting deployment by two large hosting providers, SecureSites on August 7/8 and Verio on August 12/13.

Figure 8 slightly overestimates the number of patches that were applied, since only servers which answered and were vulnerable were counted. However, this effect is slight—during the measurement period no more than 35 servers were unavailable at any given time.

5.3 Day-by-day results

It's natural to wonder whether administrators are more likely to upgrade on one day than another. In order to explore this, we first need to control for the overall upgrading trendline, which is an exponential. In order to do so, we take the daily rate of change in vulnerable hosts as a percentage of the number of vulnerable hosts. So, for instance, if we have two samples ($time_0, hosts_0$) and ($time_1, hosts_1$), the normalized rate of change over that period is given by:

$$d\frac{hosts}{dt} = \frac{hosts_1 - hosts_0}{(time_1 - time_0) \cdot hosts_0}$$

Because our samples are over 24 periods, we now have the normalized daily rate of change. Samples are taken at 6:00 AM Pacific and so we arbitrarily assign the rate of change between day X and day $X + 1$ to day X. This produces some distortion because the assigned days do not exactly match days in North America, but of course this would be true of any dividing line we chose. Realistically, not much happens in North America before 9:00 AM Eastern so this dividing line produces a relatively accurate picture of what happens each day. Alternately, one can think of days as being measured in a time zone somewhere in the middle of the Pacific. The result is shown in Figure 9 as a scatter plot. Since the results span many weeks there are a large number of points corresponding to each day of the week. The horizontal line shows the average rate of change for each day.

As Figure 9 shows, qualitatively more upgrading happens during the North American workweek than on the weekends. There is probably some effect here from the release time of the advisories (Tuesday morning and Friday morning), because we would expect the first day or two afterwards to be especially active. However, the general trend of all the data points is higher during the week than on the weekend. Note that this is a qualitative result only.

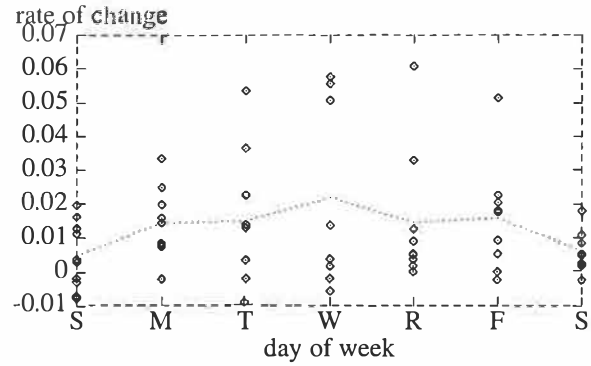


Figure 9 Daily rates of change

With the exception of the two mass upgrades mentioned in Section 5.2, there does not appear to be any significant difference in the amount of upgrading that happens during the North American daytime and the North American evening. However, it's possible that our measurements are not sensitive enough to distinguish evening from morning due to the sampling frequency and times.

5.4 Who upgrades?

Finally, let's consider the question of which class of users choose to deploy fixes. This question is not of completely academic interest. It would be convenient for vendors (if not altogether to the public good) if some readily identifiable group of users deployed and the rest did not, since they could then either ignore or cater to specific segments. (This information would be useful to attackers as well, one supposes).

In this section, we consider a number of potential predictors of administrator response. In general, we might expect that a server which is actively maintained would be more likely to be updated in response to security problems. We therefore consider a number of measures of how actively maintained the server is, including whether software versions were up to date, whether it is hosted by a hosted service provider (HSP)—which presumably maintains it more actively—and a number of metrics of how "live" the web site is. We also wanted to assess the effect of top level domain (TLD) (reported to be relevant by Moore et al. [17]).

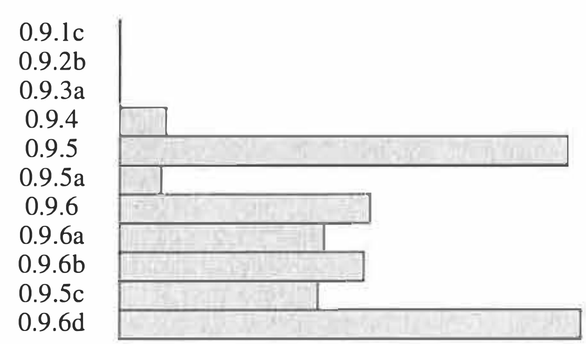


Figure 10 Fix deployment by original version

Version Effects

The intuitively obvious predictor of whether users will apply fixes is whether they already are up to date. Figure 10 shows the percentage of servers initially running each version who applied fixes to their servers by the end of the initial 28-day window. With the exception of servers initially running version 0.9.5, we see a relatively consistent pattern: servers which were initially up to date are fixed at the highest rate (70%). Servers which are running other 0.9.6 versions are fixed at a rate of about 30-40%. Less than 10% of servers 0.9.5 and 0.9.4 variants were fixed, and no servers running versions < 0.9.4 were fixed. The overrepresentation of servers running 0.9.5 is a sampling artifact—WHOIS lookups on the relevant IPs show all the servers to be operated by Cybergate, so their results are not independent.

Analysis bears out this qualitative assessment. The standard tool for analyzing multifactor categorical data, logistic regression [18, 19], allows us to estimate the relative contributions of individual predictors to a measured outcome variable. Figure 11 shows the results of a logistic regression predicting the occurrence of upgrading. The explanatory variables are the initial versions of OpenSSL and Apache, with the Apache version being a dichotomous variable of either up to date (post the chunked buffer overflow) or old. The regression was performed with R [20] using the Design [21] library.

Coefficient	Meaning
χ_1	OpenSSL 0.9.6x (x<d)
χ_2	OpenSSL 0.9.6d
χ_3	Apache Current
χ_4	OpenSSL 0.9.6x (x<d) * Apache Current
χ_5	OpenSSL 0.9.6d * Apache Current

Variable	Coefficient	Standard Error	Z	P value	Odds Ratio	95% CI
Intercept	-1.52	0.217	-7.03	-	-	-
χ_1	0.335	0.261	1.29	0.199	1.40	0.839-2.33
χ_2	1.52	0.740	2.06	0.040	4.58	1.07-19.5
χ_3	-0.782	0.567	-1.38	0.168	0.458	0.151-1.39
χ_4	1.82	0.602	3.02	0.003	6.14	1.89-20.0
χ_5	1.54	0.925	1.67	0.096	4.67	0.762-28.6

Figure 11 Logistic regression model for updating behavior

The first column of Figure 11 shows the predictive factors. For space reasons these are abbreviated as χ_x, as shown in the first table. The remaining columns show the fit coefficients along with their standard errors and Z scores which are used to compute the P values, indicating whether the predictor appears to be significant. Next, the odds ratio is shown, indicating the extent to which the presence of the predictor increases the likelihood of the measured outcome. Finally, the 95% confidence interval bounds for the odds ratio are shown.

As expected, we see that initial version is a significant predictor of updating behavior. Servers which were running 0.9.6d when the bug was released were more likely (odds ratio=4.58; P=0.04; 95% CI= 1.07-19.5), to respond than those running pre 0.9.6 versions, regardless of what Apache version they were running. However, the Apache version is a significant effect modifier when the initial OpenSSL version is 0.9.6x (x<d), even though Apache version by itself is not significant. We interpret this result as follows: The latest Apache versions contain fixes for a remotely exploitable hole for which exploits [2] were circulating. Thus, Apache version is a discriminator between people who just happen to be up to date and people who are consciously remaining up to date on security fixes (in this case for Apache).

Hosting Services

Unfortunately, the results obtained in the previous section are not entirely adequate. The problem is that the samples are not truly independent, since a fair number of the sites in question are operated by large hosting service providers. In general, we would expect all the machines operated by a given provider or administrator to be upgraded simultaneously, regardless of what software versions they were running. Discovering the operator of any given server would require contacting each administrator and asking relatively detailed questions. We considered this prohibitive.

Instead, we used WHOIS [22] net blocks as a proxy. We first interrogated the ARIN database. In cases where ARIN reported that the net block was administered by a different registry such as APNIC or RIPE we then interrogated that registry's database. We considered that any servers which belonged to the same net block were operated by the same provider.

This assignment is necessarily somewhat rough. First, some providers do not report net blocks which are assigned to customers as delegated. The result will be that some hosts which are actually operated by different administrators will appear to be operated by the same provider. Second, some large ISPs (e.g. Verio) own multiple net blocks with different names, perhaps due to acquisitions. We made no attempt to determine whether the machines in these blocks were separately administered. Finally, some providers provide both managed and unmanaged service, in which case some machines in a managed facility may actually be operated by a customer.

Nevertheless, there is quite a noticeable effect of HSP size on updating behavior. Figure 12 shows the vulnerability percentage for some somewhat arbitrarily chosen intervals of HSP sizes.

HSP Size (Hosts)	% Vulnerable	# Hosts
1-4	0.71	673
5-15	0.50	46
15-30	0.33	69
30-100 (Verio)	0.15	69

Note: only hosts for which data was available at day 27 are listed here.

Figure 12 Upgrading by HSP size

We attempted to directly account for the effect of HSP size via logistic regression but were not able to achieve good fits. Reduced models did not have satisfactory fit statistics and the saturated model produces absurd coefficients due to a number of empty cells. These problems suggest that HSPs behave enough differently from ordinary administrators that it's worth treating them separately. Accordingly, we performed a new analysis with only the "independent" hosts—those with *HSPSIZE* = 1 (n=518). This yields an excellent fit, as shown in Figure 13

With the effect of HSPs removed, we now see that all three predictors are significant. In other words, having an up to date or even modestly recent version of OpenSSL and Apache was a significant predictor that the server will be updated to fix this bug. Note that we do not show interaction terms between OpenSSL and Apache versions in this fit. Likelihood Ratio testing indicated that those terms did not significantly improve the quality of the fit. The results shown in Figure 13 are quite robust with regard to our classification of whether the server is operated by an HSP. We achieved good fits (P=.74) even when the breakpoint was set as high as 15 servers in a net block.

By contrast, attempts to fit the behavior of HSP-hosted servers (with either breakpoint 1 or 15) were unsuccessful, for three reasons. First, since HSPs were comparatively responsive, there's less variation to account for and what was seen wasn't consistent across the factors under study, as shown in the contingency table in Figure 14. Second, the number of hosts in this stratum was somewhat smaller (n=373). As a consequence of these two factors, there were a number of empty cells in the contingency table, leading to misleadingly high coefficients. Finally, a single HSP (Alabanza) failed to respond at all by day 27. We conclude that the primary source of inter-HSP variation is some factor which is not tightly related to deployed software version.

Variable	Coefficient	Standard Error	Z	P value	Odds Ratio	95% CI
Intercept	-2.40	0.325	-7.38	-	-	-
x_1	1.04	0.350	2.96	0.003	2.82	1.42-5.60
x_2	1.49	0.445	3.35	0.001	4.44	1.86-10.6
x_3	0.873	0.244	3.57	0.000	2.39	1.48-3.86

Hosmer-le Cessie Test: P=.56

Figure 13 Updating behavior of independent servers

	OpenSSL Status		
Apache Status	Downrev	0.9.6x (x<d)	0.9.6d
Downrev	.61	.71	.5
Up-to-date	1	.41	.18

Figure 14 HSP vulnerability by Apache and OpenSSL version

The Effect of TLD

Moore et al. [17] report that TLD was a significant predictor of upgrading behavior in response to the Code Red worm. In order to test for this effect we took our day-27 results and filtered out all servers corresponding to TLDs with n<10. The remaining TLDs were: .au, .ca, .com, .de, .edu, .gov, .net, .org., and .uk) The chi-squared test provides no evidence of this effect ($\chi^2 = 2.665$; df=7; p=0.914). However, the sample size in that study was larger and therefore may have more resolving power.

Some Other Irrelevant Factors

In the search for other predictors, we also examined a number of factors. These included:

- Certificate status (valid, self-signed, etc.) — as a predictor of whether the site was "real" or not.
- Website "last-modified date" (only available for 418 sites) — as a predictor of whether the site was being actively maintained.

Neither factor showed any significant predictive value.

5.5 Upgrade or Patch?

Versions prior to the current version (downrev versions) are a perrenial source of difficulty for software vendors. Even for vendors, such as the OpenSSL group, who do not make money off of upgrades, a substantial amount of time is spent providing support for older versions. Accordingly, vendors would very much like users to upgrade. Users, naturally, resist upgrading because it is disruptive. Upgrading to fix this kind of security problem, which is inherently a very small change to the code, is especially painful if it requires installing a completely new version with the attendant problems of interface instability and new bugs.

Since the OpenSSL team released both patches and a new version, an obvious question to ask is: did users primarily patch their existing code or did they primarily install a new version? Figure 15 shows the installation of both patches and upgrades during the period under study.

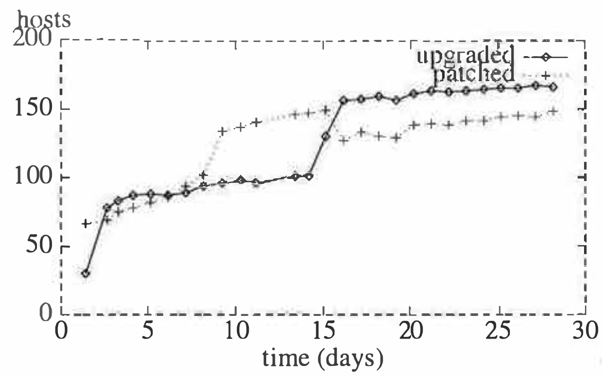


Figure 15 Propagation of upgrades and patches

As Figure 15 shows, patching and upgrading are about equally popular. This isn't surprising since, as described in Section 3.4, a large number of vendors shipped patches to the OpenSSL versions they already shipped rather than upgrades to 0.9.6e.

The data shown in Figure 15 has three additional interesting features. First, in the day immediately after the announcement, vastly more users chose to patch than upgrade. We suspect two causes for this behavior. First, the patches were released an hour or two earlier than the 0.9.6e and so a really diligent administrator might have chosen to protect himself when only the patches were available. Our data is not fine-grained enough to test this hypothesis. The second possible cause is that administrators were in a hurry to deploy a stopgap fix and that patches were easier to deploy because they were less disruptive.

This hypothesis is somewhat borne out by the data between days 12 and 15 in which the number of fixed servers upgraded servers increased while the number of patched servers declined. What happened here is that a number of administrators first deployed patches and later upgraded to 0.9.6e.

In order to study this issue, we again built a logistic model for the patch/upgrade choice (using Upgrade as the event of interest). As before, we first stratified the data on *HSPSIZE* = 1. This allowed us to obtain an adequate fit for independent servers, which is shown in Figure 16. Note that the model here does not include an interaction between the OpenSSL version and Apache version since an adequate fit was obtained without that term.

Event: Upgrade

Variable	Coefficient	Standard Error	Z	P value	Odds Ratio	95% CI
Intercept	-0.250	0.650	-0.38	-	-	-
x_1	-1.28	0.698	-1.84	0.066	0.278	0.071-1.09
x_2	-0.166	0.818	-0.20	0.839	0.847	0.171-4.21
x_3	1.26	0.484	2.61	0.009	3.53	1.37-9.13

Hosmer-le Cessie Test: P=0.92

Figure 16 Logistic regression model for upgrade/patch decision

The results here are relatively easy to interpret: users who are up to date on Apache are more likely to upgrade than apply patches. OpenSSL version is not a significant predictor. It's not clear why Apache version would be an important predictor when OpenSSL version is not. One possibility is that some subset of users respond to security problems by upgrading to the latest version. That subset would obviously have up to date versions of Apache and then would subsequently upgrade their OpenSSL versions in response to a vulnerability. One test of this hypothesis would be to measure the versions of other software deployed on the machines in our sample.

We can see a similar situation with HSPs. Of the 193 servers with *HSPSIZE* > 1 that responded to this vulnerability, 113 (58%) upgraded to newer versions of OpenSSL. There were too many empty cells to perform an adequate logistic fit but Figure 17 tells the story clearly enough.Qualitatively, servers with downrev versions of Apache were more likely to be running downrev versions of OpenSSL and more likely to patch. Servers with uprev versions of Apache were more likely to be running OpenSSL 0.9.6 and more likely to upgrade. Servers running OpenSSL 0.9.6d were more likely to upgrade. There is no reason to doubt the obvious interpretation here—it's vastly easier to upgrade if you're already more or less up to date.

Apache	Downrev			Up-to-date		
OpenSSL	Downrev	0.9.6x (x<d)	0.9.6d	Downrev	0.9.6x (x<d)	0.9.6d
Patched	19	21	0	0	31	9
Upgraded	0	2	4	0	22	55

Figure 17 Upgrade/patch decision for HSPs

5.6 SSLv2 Not Disabled

Note that we have not discussed hosts which disabled SSLv2. At no time during the selected period did more than 10 servers which supported SSLv3 disable SSLv2 and only 5 consistently appeared to have done so. It's quite possible that the remainder of hosts which appeared to have disabled SSLv2 were merely attributable to measurement error.

This is a surprising result because this countermeasure is extremely easy to deploy and completely removed the vulnerability, intuitively one would expect administrators looking for a stopgap to employ it first and only later upgrade. However, essentially no administrators did so. This could result from a number of causes, but we believe that the most likely one is simply that administrators didn't understand that it was available. Although the original advisory mentioned that only SSLv2-capable systems were vulnerable, a large number (90+ %) of the vendor advisories

did not. Moreover, as described in Section 2.2, there was widespread confusion about the scope of the problem, even among security experts who had read the original advisory. Therefore, it seems likely that administrators may simply have believed that fixing their implementation was their only option. We have no way of directly testing this hypothesis with our current data set. However, if a similar vulnerability arises with an immediate workaround but no fix we could examine how many users apply the workaround.

5.7 Deployment of Version 0.9.6g

As described in Section 2.3, the rush to deploy fixes for the initial round of bugs resulted in an incomplete fix with some DoS vulnerabilities. The fixes for these vulnerabilities were deployed in version 0.9.6g. However, deployment of version 0.9.6g was spotty. At day 27, less than 5% of the servers studied were version 0.9.6g. At the end of the study period, 173 (19%) of the servers were 0.9.6g and 121 (14%) were 0.9.6e. It is not completely clear why this is the case, but there were likely several contributing factors:

- The announcement was not as widely disseminated as the original announcement.
- The problems were not as serious (DoS only).
- Not all vendors released updates for the problems discovered in the first round of patches.

In general, it appears that users who upgraded to 0.9.6e did not upgrade again to 0.9.6g.

5.8 Response to Worm Announcement

The announcement of the Slapper worm initiated a new round of upgrades and patches. Figure 18 shows the data for the first two weeks after the announcement of the worm. As before, we have trimmed the graph to show the relevant portions of the data and fitted an exponential ($r^2 = .99$).

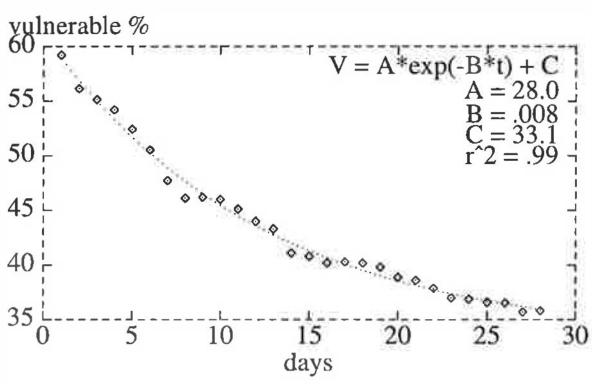


Figure 18 Upgrading after Slapper announcement

This second round of updates actually requires some explanation. Why do people who didn't upgrade the first time upgrade when the worm was announced? One possibility is simply that some administrators had not heard of the original bug. However, since the worm was announced through the same channels, this doesn't seem likely to account for the entire difference. A more likely explanation is that administrators have explicitly or implicitly adopted a strategy of only upgrading when an exploit is available, rather than merely when a bug is announced. As Beattie et al. [23] point out, it's sometimes undesirable to update immediately, since patches are flawed. Moreover, it may not be convenient to patch immediately. Waiting until an exploit has been released therefore seems like a reasonable wait-and-see attitude.

In order to study this, we built a logistic regression model. As before, we focused on independent sites by removing all sites from net blocks containing more than one host. The results are shown in Figure 19.[6]

Variable	Coefficient	Standard Error	Z	P value	Odds Ratio	95% CI
Intercept	-1.358	0.256	-5.31	-	-	-
x_1	0.647	0.296	2.18	0.029	1.93	1.07-3.45
x_2	0.261	0.484	0.539	0.590	1.30	0.502-3.36
x_3	0.897	0.270	3.32	0.001	2.45	1.44-4.16

Hosmer-le Cessie Test: P=.13

Figure 19 Logistic regression model for post-worm updating

As Figure 19 shows, OpenSSL 0.9.6x (x<d) is a significant but small predictor for post-worm response (odds ratio=1.93; P = 0.029; 95% CI=1.07-3.41). Apache up-to-date status is a somewhat better predictor (odds ratio=2.45; P=0.000; 95% CI=1.44-4.16).

Note, however, that OpenSSL 0.9.6d is not a significant predictor. We interpret this result as follows: any administrator who was staying up to date on OpenSSL updated when the bug was announced. The remaining administrators respond to exploitable bugs. Since Apache status is a signal for whether or not administrators respond to public exploits, it therefore is a factor in predicting whether or not administrators will upgrade after release of the worm.

The implication of this analysis is that there are three roughly equal-sized classes of users.

- Users who respond immediately upon the release of security holes.
- Users who respond when exploits for holes start to circulate.
- Users who do not respond at all.

In order to confirm this hypothesis, one would need to do a horizontal study of updating behavior for a wide number of independent packages with variation in the known types of security flaw. A (non-random) sample could potentially be

found by examining other software present on the systems in this sample.

Finally, we turn to the behavior of HSPs subsequent to the release of the worm. As mentioned previously, HSPs were in general very responsive to the initial disclosure of this bug. The single HSP that did not respond at all to the initial disclosure (Alabanza) responded completely upon release of the worm. This single HSP has a large effect on our estimates of which variables are significant. No matter how the data is analyzed, Apache version remains a strong predictor, but we are unable to draw firm conclusions about the effect of OpenSSL version.

6 Policy Implications

Since this paper reports on a single disclosure event, we should be wary of drawing firm conclusions. Nevertheless, if we take this event as representative, administrator behavior in this case suggests some appropriate methods for managing future vulnerabilities.

6.1 Timing of Disclosure

As we saw in Section 5.2, the deployment of fixes is very strongly frontloaded. In both rounds of upgrading, nearly all of the administrators who eventually fixed their servers had done so within three weeks of the announcement of the vulnerability. However, a substantial fraction of those administrators did not upgrade within the first week. These two results have implications for the timing of bug disclosure.

Delayed Full Disclosure

Until recently it used to be quite common for vendors and coordination centers to provide a limited advisory that described the nature of the vulnerability in general terms and notified users of the availability of a fix. In some cases the vulnerability was never fully disclosed. In others, the vendors would wait a "decent interval" to allow users to upgrade. We are now in a position to assess this practice.

Since many users never upgrade we can easily see that it is never "safe" to release all the details of a vulnerability. There will always be a large number of users who remain vulnerable at any given time. However, it is similarly clear that a long delay between limited disclosure and full disclosure serves no useful purpose. As essentially all users who are going to upgrade do so quickly, full disclosure should take place within a month or not at all.

We take no position on the relative merits of full disclosure versus limited disclosure in general, except to note the obvious fact that full disclosure necessarily leaves a large number of users vulnerable. On the other hand, it is often argued that full disclosure encourages users to upgrade, which is no doubt a good thing. Whether full disclosure has benefits that outweigh these costs is not a matter for discussion here. Obviously, this tension is far weaker in the case of Open Source software, since the deployment of patches almost inevitably provides enough information for an attacker to discover the nature of the vulnerability.

Disclosure Before Fixes Are Available

As we saw in Section 5.2, the users who are going to upgrade are very quick to do so. As a consequence, disclosing vulnerabilities before fixes are available results in a very high marginal cost to those users, since those users are vulnerable for the period between announcement and release of the patch—whereas if disclosure were withheld they would not be. The negligible use of non-fix countermeasures seen in Section 5.6 suggests that the availability of workarounds does not reduce this marginal cost. If at all possible, vendors should be given time to develop and deploy fixes.

6.2 What Fixes to Develop

Obviously, vendors would prefer to minimize the number of versions they support. In the best case, all users would upgrade to the latest version. Even if this is not possible, it would be desirable to issue patches only for the most recent versions. Unfortunately, as we saw in Section 5.5, a substantial fraction of users across all versions preferred patches to upgrading. We don't have enough data to determine whether or not those users would have upgraded were patches not available, but it seems likely that many would not have. By contrast, the value of devising and deploying workarounds seems quite minimal, given that essentially no users chose to deploy even the most minimal form of workaround.

6.3 Get it Right the First Time

It is universally agreed that OpenSSL 0.9.6g is superior to 0.9.6e. Nevertheless, at day 27, deployment of 0.9.6e was vastly greater than 0.9.6f/g for the simple reason that 0.9.6e got there first and solved the most pressing security problem: the remote buffer overflow. However, had 0.9.6g been released instead of 0.9.6e the DoS problems would be fixed as well. The rush to release 0.9.6e is more than understandable in view of the seriousness of the bugs. Nevertheless, the result was not quite optimal. Note, however, that users who upgraded after the worm was announced naturally installed 0.9.6g. The important lesson here is that users have a limited appetite for security fixes and that vendors who wish to deploy all their fixes should ensure that their first bugfix release includes them all.

6.4 The Impact of Vulnerable Servers

The impact of Web server compromise is obviously quite substantial. Aside from the high probability that the attacker will leverage the compromise into administrator privileges, mere control of the server itself is dangerous. The attacker can vandalize the site, or, worse yet, recover the server's private key. The lack of reasonable revocation mechanisms means that private key compromise is catastrophically bad for the site in question. A compromised key allows the attacker to impersonate the server as well as to passively decrypt any connections he can sniff using a tool such as ssldump [24]. The private key is generally stored in the clear in the server's memory and so recovering the private key once the server is compromised is trivial and does not require achieving root access.

One common question that people seem to have is whether the residually vulnerable servers may not simply be defunct. This does not seem to be the case. Although we have not done a thorough survey, the list of vulnerable servers as of this writing includes a number of "prestige" sites, including universities, government agencies, and well known ISPs. One interesting avenue of research would be to correlate blind user ratings of site "liveness" against vulnerability.

Even if the vulnerable sites were defunct, this would still be a security problem for the rest of the network, because compromised machines can be used as an attack platform. Recall that the Slapper worm does exactly this, turning the victim machine into a zombie for mounting DDoS attacks. Thus, it's critical for everyone for vulnerable machines to be fixed. This risk is particularly great, since, as shown in Section 5.4, independently operated servers are less likely to upgrade, and such servers are the most difficult for victim sites to have shut down. Potential measures along these lines would be for ISPs to filter un-patched servers. Dick [25] suggests a cleverer though perhaps more difficult option, which is for sites likely to be vulnerable to DDoS to pay other sites to upgrade.

6.5 Understanding HSP Policy

As we have seen, servers operated by HSPs generally are more responsive than independent servers. However, we have also seen that not all HSPs are equally responsive. Accordingly, users who care about security should investigate their HSP's upgrading policy and (if appropriate) insist that the HSP provide response guarantees.

7 Sources of Error

In any survey of this type, there are a number of potential source of error. This section describes the known sources of error and attempts to assess their severity.

7.1 Availability Bias

Our sampling procedure is not guaranteed to give a completely random sample. In particular, it favors sites which advertise in Google. This creates two known sorts of bias:

- Bias against private sites which do not advertise.
- Bias against large sites which restrict searching.

These forms of bias act in opposition. The first eliminates some small sites which are unreachable via links. One might hypothesize that such sites are less well maintained and therefore less likely to respond to security vulnerabilities. The second form of bias eliminates sites which one might hypothesize were well managed and therefore more likely to respond. The current data do not allow us to evaluate the extent of either form of bias.

The extent to which this sort of sampling bias influences our results depends on what sort of conclusions you wish to draw. If one wishes to draw conclusions about the probability that a web site one accesses will have been compromised, this form of sampling is good. If one wishes to estimate the percentage of potential "zombie" machines available for DDoS, then it is possible that our sample is misleading. One way to crosscheck would be to sample IPs randomly as common worms do. We have not done so.

7.2 Network Problems

This sort of experiment is subject to two forms of network failure. In the first, the network connection to our sampling machine failed and we were therefore unable to take data during the outage. In such cases, we resampled after the outage. No such outage lasted more than 4 hours and so we do not believe that this has an appreciable effect on our data.

The second sort of outage, inability to reach a specific server, is more troubling, since there is no reason to believe that unavailability of hosts is independent of other properties. However, since no more than 6% of hosts became unavailable during the sample period, the bias is relatively small compared to the size of the effects we're measuring.

7.3 Measurement Error

Remote probing is subject to several forms of measurement error:

- False version numbers.
- Load balancing.
- Acceleration

False version numbers

It's possible, though inconvenient, to modify one's mod_SSL installation to advertise a false version number. Thus, one could (for instance) advertise OpenSSL 0.9.6e while actually running OpenSSL 0.9.6d. We do not believe that this is widely done. Anyone with the expertise to make such modifications could quite easily apply patches.

Another form of false version numbers derives from the fact that the advertised version number is determined from the OpenSSL header files and is thus set at compilation time. If the OpenSSL shared library is replaced without recompiling mod_SSL, there will be a mismatch between the advertised and actual version numbers. Generally, this effect will make the OpenSSL version seem older than it actually is, since it is uncommon to replace one's library with an older version.

The impact of this sort of error has not been completely determined. However, we have developed some prototype techniques for remotely fingerprinting OpenSSL versions based on small behavioral differences. Our preliminary results suggest that <5% of servers are advertising the wrong version. In the future we hope to develop improved fingerprinting technology and thereby determine versions more precisely.

Moreover, since we're primarily concerned with predicting server behavior from externally visible behavior, thus most of our results—with the exception of the discussion of patch versus upgrade in Section 5.5—are unchanged when interpreted in terms of advertised version instead of true version.[7]

Load Balancing

Suppose an operator runs more than one server behind a load balancer. For some reason or other, one machine is fixed and the other is not. In this case, we will get inconsistent results depending on the load balancer. In our tests, about 10 machines had repeated flip-flops between fixed and unfixed states. Note that this suggests that at least in these cases, not all machines on a given site had been upgraded simultaneously.

Accelerating Proxies

It's quite common to use an SSL reverse proxy [26] to offload SSL processing from a server. However, since the proxy is otherwise transparent, this means that the advertised OpenSSL version (from the server) might not match the real SSL implementation (whatever is on the proxy). Since most such proxies are based on OpenSSL, this might mean the proxy was vulnerable even though the server advertised a non-vulnerable version. Our data shows a small number of hosts (<10) which advertise non-vulnerable versions of OpenSSL but appear under probing to be vulnerable. Careful evaluation of the OpenSSL code indicates that the probes must generate an error with any repaired SSL version. Therefore, we believe that this effect is responsible for the anomalies in question.

Block Ciphers versus Stream Ciphers

Subsequent to the survey we discovered a possible source of false negatives—`hstest` works by using a key-arg one byte longer than it should be. This causes an overrun when a block cipher is used but not when a stream cipher is used. Thus, if we negotiate a stream cipher, then the handshake will succeed. However, this is not a significant source of false negatives, for two reasons:

1. In SSLv2 the client chooses the cipher suite from a list of those acceptable to the server. Our client prefers block ciphers.
2. Nearly all servers support at least one block cipher.

Double-checking with a client programmed to support only block ciphers yields essentially identical results (+/- 1 server).

7.4 Analytical Issues

Sample Size

The relatively small sample size (n=891) presented some analytical challenges. In some cases, we were unable to fully fit parameters of interest because the corresponding cells in the contingency table were empty, leading to absurd results. A larger sample would presumably have fewer completely empty cells and be more amenable to analysis.

Server Independence

As noted in Section 5.4, not all of our samples are completely independent. We adjusted for this by stratifying the sample into "independent servers" and HSPs. However, this stratification is necessarily somewhat imprecise. However, the fact that which predictors are significant is relatively insensitive to where the exact boundary is placed suggests that the stratification is good enough to draw reasonable conclusions about relevant factors. Confirmation of these results could be achieved by directly contacting server administrators and determining exactly who is responsible for which hosts.

8 Related Work

The general shape of the upgrading curve has been observed by previous authors. The closest related work is by Provos and Honeyman [27] who measured the deployment of OpenSSH versions subsequent to the release of the SSH

CRC vulnerability [28]. The authors found a somewhat lower asymptotic (20%) vulnerability rate, but it's not clear how reliable it is since in some cases they notified the hosts under study. Cheswick et al. [29] report similar results for response to a vulnerability in BIND. Moore et al. [17] report on user upgrading behavior in response to the Code-Red worm. Finally, Browne et al. [30] measured the rate of intrusions resulting from known bugs, thus implicitly measuring the rate of patching.

This work differs from previous work in three important respects. First, we follow the trajectory of a defined set of hosts from release of a bug through the identifiable release of a worm that exploits the bug. Previous work only measured one phase or the other of upgrading. This allows us to accurately compare response to different security stimuli. Second, because we are directly probing for vulnerabilities rather than merely version numbers, we are able to determine when servers have been patched or countermeasures have been applied as well as whether they have been upgraded. As Section 5.5 shows, a significant number of administrators opt to patch and therefore it's important to measure patch rate when patches are available.

Finally, since this is a longitudinal study of a specific set of hosts, we are able to characterize the various factors that predict administrator responses. We are not aware of previous work that addresses this issue.

	Initial Release		Post Worm	Upgrade/Patch
Factor	Overall	Independent	Independent	Independent
OpenSSL < 0.9.6d (x_1)	1.40	2.82	2.45	N/A
OpenSSL 0.9.6d (x_2)	4.58	4.44	2.0[illegible]	N/A
Apache Up-to-date (x_3)	-	2.39	-	3.53
$x_1 * x_3$	6.14	N/A	-	-
$x_2 * x_3$	-	N/A	-	-

Figure 20 Summary of significant predictors

9 Conclusions

We have described a longitudinal study of user response to security flaws. Our study begins with the announcement of the vulnerability and continues through the release of a worm that exploits that vulnerability. We report two primary results, one regarding the rate of user response and the second regarding the composition of the responding groups.

We first observe that administrator response is poor. Three weeks after the initial vulnerability announcement, fix deployment had leveled off and 60% of the sample was still vulnerable at 27 days. Data collected in the three weeks after the release of the Slapper worm showed a similar pattern with a projected asymptote of 32% vulnerable.

We also identified a number of predictors for server behavior, summarized in Figure 20. In the first round of upgrading, servers running recent versions of OpenSSL and Apache were overrepresented in the responding population. In the second round of upgrading, servers running recent versions of Apache were overrepresented. Even after two rounds of upgrading, a fair number of servers running relatively recent OpenSSL distributions remained vulnerable, including several "prestige" servers. In general, large hosting service providers appear to be rather more responsive. Roughly twice as high a percentage of HSPs upgraded in the first round as compared to the population average.

Acknowledgments

The author would like to thank the ICIR Wednesday Seminar, in particular Sally Floyd, Mark Handley, Vern Paxson, and Scott Shenker, for their comments on the talk that turned into this paper. Jeff Schiller warned me about the bug prior to its release, allowing me to get my probes in place. Kevin Dick, Lisa Dusseault, Allison Mankin and Terence Spies provided discussions about experimental technique and direction. Phil Beineke provided assistance with the statistical analysis. Thanks are also due to Frank Harrell and the members of the R-help mailing list for assistance with R. Solar Designer pointed out the possibility that advertised version numbers might not match real versions. Mark Schertler provided comments on the presentation. Thanks also the anonymous reviewers from the USENIX Security Program Committee.

Notes

1. 25% of servers upgraded is a lower bound on fix deployment since a variety of countermeasures and patches for the Apache flaws were available.
2. SSLv2-only servers have taken quite a long time to disappear, because a number of Netscape Enterprise server versions were SSLv2-only. This fact may explain the widespread concern over SSLv2-only clients.
3. Our probes appear to intrusion detection systems as attempts to exploit the bug. In addition, a four times a day probe is noticeable to a small site.
4. A similar technique was described by Weimer [31] while this paper was in production. Weimer appears to have developed his technique based on analysis of the Slapper worm.
5. In addition, we were unable to fit the saturated model because there were no servers with (OpenSSL 0.9.6d and Apache downrev).
6. The fit here is not particularly good, as shown by the low P value (P = .13). However, the saturated model does not have a significantly better log likelihood (P > .10) and none of the interaction terms are significant. However, as before, we find that the fit parameters are stable when we adjust the

stratification point. In fact, if we use 15 as our cutoff, the parameters are qualitatively similar but the fit is far better (P = .94).

7. We are only interested in dividing versions into broad classes, not (for instance) in the difference between 0.9.5 and 0.9.5a

References

[1] Netcraft, *Web Server Survey Home Page.* `http://www.netcraft.com/`

[2] CERT, "Apache Web Server Chunk Handling Vulnerability," CERT Advisory CA-2002-17 (June 17, 2002). `http://www.cert.org/advisories/CA-2002-17.html`

[3] Nunes, F., "bugtraq.c httpd apache ssl attack," Bugtraq posting (September 10, 2002.).

[4] Laurie, B., *"OpenSSL Security Alert - Remote Buffer Overflows",* OpenSSL Mailing List (August 2002).

[5] Akkerman, W., Debian Security Advisory DSA-136-1 (July 30, 2002).

[6] *Trustix Secure Linux Advisory 2002-0063* (July 29, 2002).

[7] *EnGarde Secure Linux Security Advisory ESA-20020730-019* (July 30, 2002).

[8] Ahlberg, D., *Gentoo Linux Security Announcement* (July 7, 2002).

[9] Slashdot, *OpenSSL Security Update.* `http://developers.slashdot.org/article.pl?sid=02/07/30/1323226&mode=thread&tid=128`

[10] *FreeBSD Security Advisory FreeBSD-SA-02:33.openssl* (July 31, 2002).

[11] *Apple Security Update 2002-08-02* (August 2, 2002).

[12] *NetBSD Security Advisory 2002-009* (September 22, 2002).

[13] Benham, M., "IE SSL Vulnerability," Bugtraq posting (August 5, 2002).

[14] OpenPKG, OpenPKG-SA-2002.008 (July 30, 2002).

[15] Hittel, S., *Modap OpenSSL Worm Analysis.* `http://analyzer.securityfocus.com/alerts/020916-Analysis-Modap.pdf`

[16] Murray, E., *SSL Server Security Survey* (July 31, 2000). `http://www.lne.com/ericm/papers/ssl_servers.html`

[17] Moore, D., Shannon, C.,, and Claffy, K., "Code-Red: a case study on the spread and victims of an Internet worm," *Internet Measurement Workshop* (2002).

[18] Kleinbaum, D., and Klein, M., *Logistic Regression: A Self-Learning Text, 2ed,* Springer-Verlag, New York (2002).

[19] Hosmer, D., and Lemeshow, S., *Applied Logistic Regression, 2ed,* Wiley, New York (2000).

[20] Ihaka, R., and Gentleman, R., "R: A Language for Data Analysis and Graphics," *Journal of Computational and Graphical Statistics.*

[21] Harrell, F. E., *Design Library.* `http://hesweb1.med.virginia.edu/biostat/s/Design.html`

[22] Harrenstien, K., Stahl, M. K., and Feinler, E. J., "NICNAME/WHOIS," RFC 954.

[23] Beattie, S., Arnold, S., Cowan, C., Wagle, C., Wright, C., and Shostack, A., "Timing the Application of Security Patches for Optimal Uptime," *Proceedings of LISA XVI* (2002).

[24] Rescorla, E., *ssldump.* `http://www.rtfm.com/ssldump`

[25] Dick, K., *Personal communication..*

[26] SonicWall, *High Availability Options for SonicWALL SSL Devices.*

[27] Provos, N., and Honeyman, P., "ScanSSH - Scanning the Internet for SSH Servers," *16th USENIX Systems Administration Conference (LISA)* (2001).

[28] Zalewski, M., "Remote Vulnerability in SSH daemon crc32 compensation attack detector," RAZOR Bindview Advisory CAN-2001-0144 (2001).

[29] Cheswick, W., Bellovin, S., and Rubin, A., *Firewalls and Internet Security, 2nd edition,* Addison-Wesley (2003).

[30] Browne, H. K., Arbaugh, W. A., McHugh, J., and Fithen, W. L., "A Trend Analysis of Exploitations," *University of Maryland and CMU Technical Reports* (2000).

[31] Weimer, F., "Remote detection of vulnerable OpenSSL versions," Bugtraq posting (September 10, 2002).

PointGuard™: Protecting Pointers From Buffer Overflow Vulnerabilities

Crispin Cowan, Steve Beattie, John Johansen and Perry Wagle

Immunix, Inc. http://wirex.com/
`crispin@immunix.com`

Abstract

Despite numerous security technologies crafted to resist buffer overflow vulnerabilities, buffer overflows continue to be the dominant form of software security vulnerability. This is because most buffer overflow defenses provide only *partial* coverage, and the attacks have adapted to exploit problems that are not well-defended, such as heap overflows. This paper presents PointGuard, a compiler technique to defend against most kinds of buffer overflows by encrypting pointers when stored in memory, and decrypting them only when loaded into CPU registers. We describe the PointGuard implementation, show that PointGuard's overhead is low when protecting real security-sensitive applications such as OpenSSL, and show that PointGuard is effective in defending against buffer overflow vulnerabilities that are not blocked by previous defenses.

1 Introduction

Despite numerous technologies designed to prevent buffer overflow vulnerabilities, the problem persists, and buffer overflows remain the dominant form of software security vulnerability. Attackers have moved from stack smashes [25] to heap overflows [5], `printf` format vulnerabilities [6], multiple free errors [1, 13] etc. which bypass existing buffer overflow defenses such as non-executable memory segments [14, 12], StackGuard [9] and libsafe [2].

All of these classes of attack work to corrupt *pointers*: Sometimes code pointers (function pointers and longjmp buffers) and sometimes data pointers [4]. In principle, an attacker can use overflows to corrupt arbitrary program objects, but in practice corrupting pointers is by far the most desirable attacker target. The reason is that the attacker is seeking *total* control of the victim process, i.e. they want the process to execute *payload* code [25] so they can get to a root privileged shell.

Thus we sought an effective defense for pointers. *PointGuard* defends against pointer corruption by encrypting pointer values while they are in memory, and decrypt them only immediately before dereferencing, i.e. just as they are loaded into registers. Attackers attempting to corrupt pointers in memory in any way can *destroy* a pointer value, but cannot produce a *predictable* pointer value in memory because they do not have the decryption key. We modify a C compiler (GCC) to emit code that encrypts pointers for storage in memory and decrypts them for dereferencing.

The rest of this paper is organized as follows.Section 2 elaborates on pointer corruption vulnerabilities. Section 3 presents the PointGuard design and implementation. Section 4 presents our compatibility testing, showing that PointGuard imposes minimal compatibility issues on commonly used software. Section 5 presents our security testing against known pointer corruption vulnerabilities in actively used software. Section 6 presents our performance testing, showing that the performance costs of PointGuard protection are minimal. Section 7 describes related work in defending against pointer corruption. Section 8 presents our conclusions.

2 Pointer Corruption Vulnerabilities

An attacker's goal in attacking a vulnerable program is to obtain that program's privileges. While the attacker could manipulate the program into directly performing

This work supported in part by DARPA contract N66001-00-C-8032.

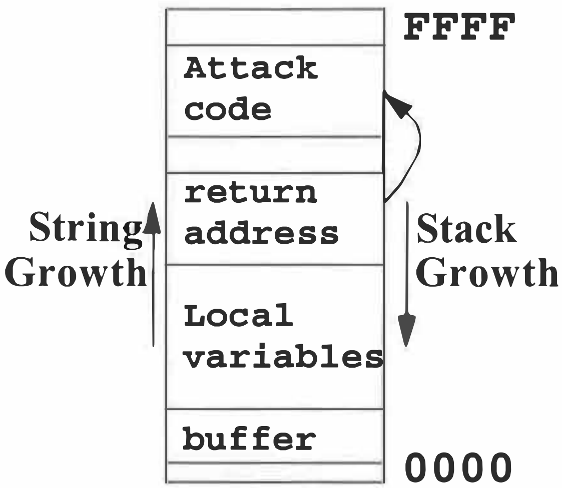


Figure 1: "Stack Smash" Attack Against Activation Record

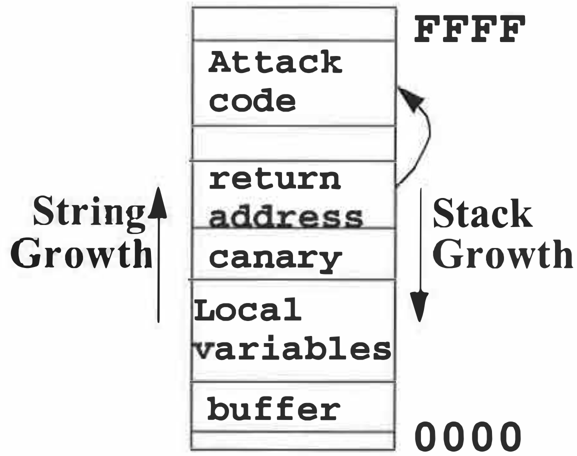


Figure 2: StackGuard Defense Against Stack Smashing Attack

some action, it is much more convenient for the attacker to get a shell prompt with the program's privileges. For these reasons, the attacker most desires to get the victim program to *execute arbitrary code*, colloquially referred to as "shell code" because the common code performs the semantic equivalent of `exec(/bin/sh)`.

To get the program to execute shell code, the attacker wants to modify a *code pointer* (a pointer that the program expects to point to legitimate code) such that the code pointer points to code the attacker would like executed. Once the code pointer has been thus modified, at some later point in execution the program dereferences the corrupted code pointer, and instead of jumping to the intended code, the program jumps to the attacker's shell code.

The classic form of buffer overflow attack is the "stack smash" described by Aleph One [25]. In this attack, buffers (character arrays) that are allocated on the stack (*automatic* variables [21] declared within the body of a C function) are overflowed with the goal of corrupting the function's return address within the activation record, as shown in Figure 1. In previous work, we provided the StackGuard defense [9] against stack smashing attacks, which detect corrupted return address values in activation records by ornamenting the activation record with a *canary* word, as shown in Figure 2. When a stack smash occurs, the overflow necessarily corrupts the canary word.

The general case of buffer overflows is to overflow buffers allocated *anywhere* (stack, heap, or static data area) corrupting whatever important state is adjacent to the overflowable buffer. Figure 3 shows a simple program subject to a static buffer overflow, where excess input intended for `statbuf` corrupts the `statptr` pointer. Figure 4 shows a similar program subject to heap overflows. In addition to threatening the adjacent buffer, this overflow also has the potential to corrupt the malloc data structures.

In principle, this would allow the attacker to induce arbitrary behavior in the victim program. In practice, the state adjacent to overflowable buffers are determined by the vagaries of data layout within the program, and depending on circumstances, may have limited semantics of use to the attacker. Thus attackers are most inter-

```
char statbuf[100];
char * statptr; // vulnerable to statbuf

myfunc() {
      statptr = malloc(1000);
      gets(statbuf);
      gets(statptr);
}
```

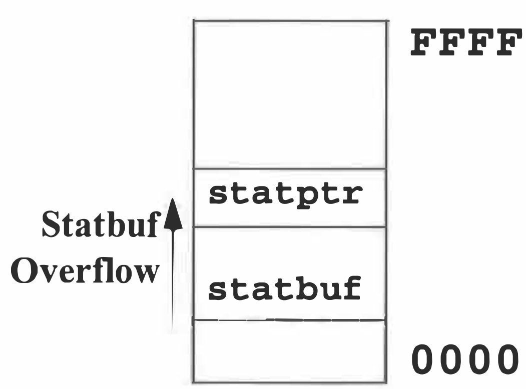


Figure 3 Vulnerable to Static Overflow

ested in corrupting *pointers* , because they give the most leverage.

Conover et al [5] describe a variety of methods to use buffer overflows against buffers located in heap and static data areas to corrupt adjacent pointers. In some cases, they directly corrupt *code pointers* (function pointers and `longjmp` buffers) to directly seize control of the victim program. In other cases, they *indirectly* use overflows to corrupt *data pointers* to point to unintended locations, and from there use those data pointers to corrupt code pointers. Thus the PointGuard defense is designed to protect all pointers from corruption.

3 PointGuard Defense Against Pointer Corruption

The PointGuard defense against pointer corruption consists of encrypting pointer values in memory and only decrypting the pointers when they are loaded into CPU registers. Section 3.1 describes basic PointGuard operations. Section 3.2 describes PointGuard encryption. Section 3.3describes our implementation of the Pointguard defense. Section 3.4 elaborates on special implementation issues that Pointguard imposes on the compiler and on developers.

3.1 PointGuard Operation

Pointers are vulnerable in memory, where they can be corrupted using various buffer overflow and printf format string attacks. Encryption protects pointers, because the attacker cannot corrupt pointers such that they will *decrypt* to a predictable value. Conversely, pointers are safe in registers, because registers are not addressable via computed addresses, and thus not subject to overflow attacks.

This scheme critically depends on pointers always being loaded into registers prior to being dereferenced. Older CPU instruction sets support various forms of *memory indirection* [23] where a pointer could be dereferenced without using a register. More recent RISC instruction set architectures dispensed with memory indirection as being too slow, adopting instead a *load/store architecture* [28] in which *all* values are loaded into registers before operating on them. Even legacy CISC instruction set architectures [17] found load/store instruction sequences to be most efficient, and compilers for these CPUs were subsequently changed to prefer load/store instruction sequences. PointGuard critically depends on a load/store instruction discipline.

Figure6 through Figure8 illustrate how PointGuard works to defend pointers. Figure 5 shows a normal pointer dereference. Figure 6 shows a normal pointer dereference under attack, where the attacker used a buffer overflow or related means to corrupt a pointer to point to a different location. Figure 7 shows a PointGuard dereference, decrypting the pointer value as it is loaded into the CPU register. Figure 8 shows a PointGuard dereference under attack. The attack fails because the attacker's corrupted value is passed through the PointGuard decryption process, producing a *random* address reference, and for reasonably sparse address spaces, probably causing the program to crash Crashing is the objective: to cause the victim program to*fail-stop* , rather than hand control over to the attacker.

Notably critical to this scheme is that the attacker cannot know or predict the encryption key. This key is a relatively easy secret to keep, because it is never shared. The key is generated at the time the process starts, using some suitable source of entropy such as reading a value from `/dev/random`. This key is then never shared with any entity outside the process's address space. To obtain the key, the attacker would either have to already have permission to manipulate the process with debug-

```
myfunc() {
        char *buf1 = (char *)malloc(BUFSIZE);
        char *buf2 = (char *)malloc(BUFSIZE);

        gets(buf1);
        gets(buf2);
}
```

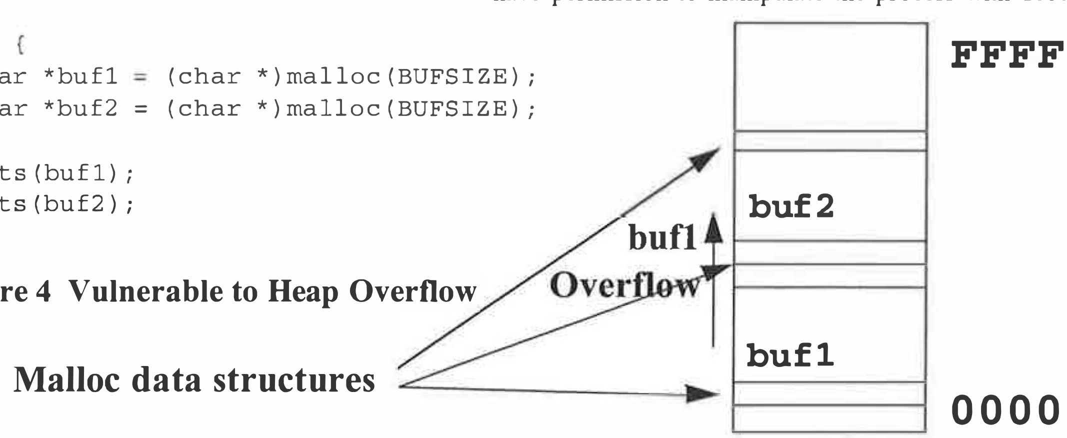


Figure 4 Vulnerable to Heap Overflow

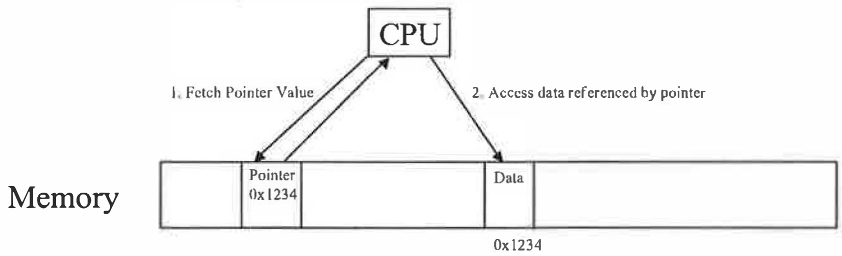


Figure 5 Normal Pointer Dereference

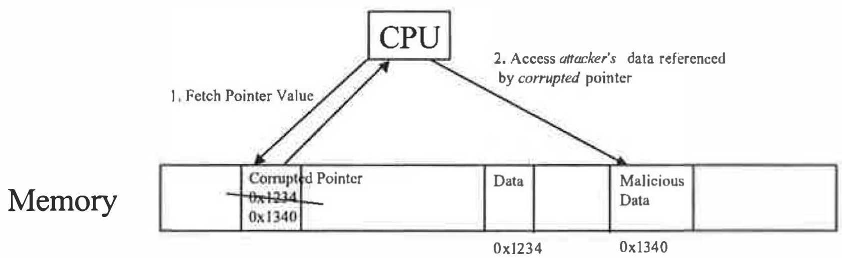


Figure 6 Normal Pointer Dereference Under Attack

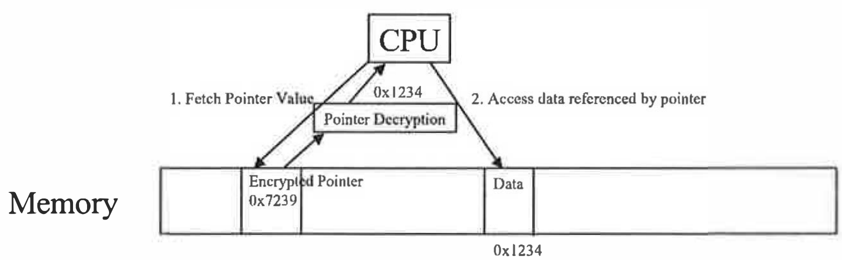


Figure 7 PointGuard Pointer Dereference

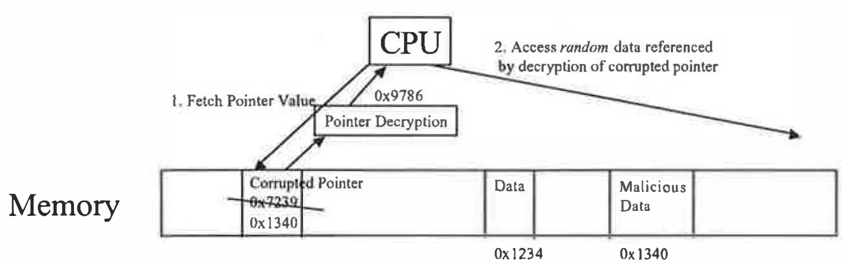


Figure 8 PointGuard Pointer Dereference Under Attack

ging tools (e.g. `ptrace`) or would have to have *already* successfully perpetrated a buffer overflow attack against the process.

The key is available to processes that share memory, but such processes are already effectively sharing enough to be fully vulnerable to each other. In fact, the PointGuard key is significantly *less* sensitive than some other, more durable objects that processes may share across a shared-memory boundary, such as file descriptors.

3.2 PointGuard Encryption

PointGuard takes the odd position of using encryption to provide integrity. PointGuard seeks to provide integrity for pointers, so that pointers cannot be modified in ways the programmer did not intend. *Encryption* provides for

confidentiality, but is cryptographically weak providing integrity, and so PointGuard would seem to be cryptographically weak.

However, PointGuard never gives the attacker a look at the ciphertext. The key is chosen anew at process exec() time, and in the absence of *already* having some way to bypass PointGuard, attackers have no way of determining what that key is, or of reading any sample encrypted pointer values. Thus the cryptographic weakness of using encryption for integrity is irrelevant, and it suffices to make it merely improbable to be able to spoof integrity in pointer values *on the first try*. PointGuard makes pointers *brittle* with respect to corruption: attackers cannot use pointer corruption to read or write any *particular* data structure.

Conversely, because PointGuard sits between the CPU level 1 cache and registers, it is very important that PointGuard be fast. There is often a few cycles of load delay slots between loading a pointer into a register value and dereferencing the pointer value, so if decryption can be done in a few cycles, it can be nearly free. Therefore, it is appropriate to use very fast and simple "encryption" techniques, which we elaborate on in Section 3.3.2.

3.3 PointGuard Implementation

PointGuard is implemented as a C compiler enhancement. PointGuard defense mechanisms are integrated into the binary programs that compiler emits. The compiler mist *consistently* perform encryption and decryption of pointers be *consistent*, i.e. that pointers are consistently encrypted at such places that they are always decrypted before use, and that only encrypted pointers are decrypted.

It is important to the security value of PointGuard that pointers are encrypted when stored in memory, i.e. storage that is addressable, and thus vulnerable to attack due to un-typed memory access. CPU registers are notably *not* addressable, and thus the ideal method is to store pointers in the clear in registers, and encrypted in memory. There are several *potential* PointGuard implementation strategies, described in Section 3.3.1. Section 3.3.2 describes our actual PointGuard implementation.

3.3.1 Potential Implementation Strategies

There are many possible places in the compiler to put the encryption/decryption of pointers. These options must all satisfy the "consistency" requirement, and trade off the security value against ease of implementation.

In the Preprocessor: It is possible to use a preprocessor (possibly C's conventional CPP preprocessor) to do a source->source translation of programs, so that all pointer expressions are transformed to include the encryption upon setting pointer values, and decryption on reading pointer values. The advantage of this approach is ease of implementation. The disadvantage is that C macro expansion has a propensity to evaluate terms in an expression multiple times, which could cause problems if it results in pointer values being decrypted *twice*. There is also a substantial risk that transient subexpressions performing the PointGuard encryption/decryption might result in temporary values containing clear text pointers being left in memory.

In the Intermediate Representation: Most compilers first transform the source code into an intermediate representation (commonly known as an *abstract syntax tree*) to perform architecture-independent manipulations on the intermediate representation, and then emit architecture-dependent instructions from the modified intermediate representation. One of the manipulations performed on the intermediate representation can be to insert code to encrypt and decrypt pointer values when they are set and read, respectively. The advantage to this method vs. the preprocessor is that it avoids the duplicate expression problem, and can be more efficient. The disadvantage to this method is that it may leave decrypted pointer values in the form of temporary terms in main memory.

In the Architecture-Dependent Representation: As above, maximal security protection is provided if pointer values are only decrypted while in the CPU's registers. However, the CPU's registers are only visible to the compiler in the architecture-dependent representation. Compilers are capable of manipulating architecture-dependent representations (e.g. for "peephole optimization") but at the cost of having to do the compiler work over again for each CPU architecture to be targeted (as in GCC, which seeks to target many CPUs, including the Intel x86, the IBM/Motorola PPC, and the Sun SPARC processors). This particular transformation would transform instructions to load pointer values from memory into registers to add the decryption, and would transform the saving of pointer values from registers to memory to add encryption.

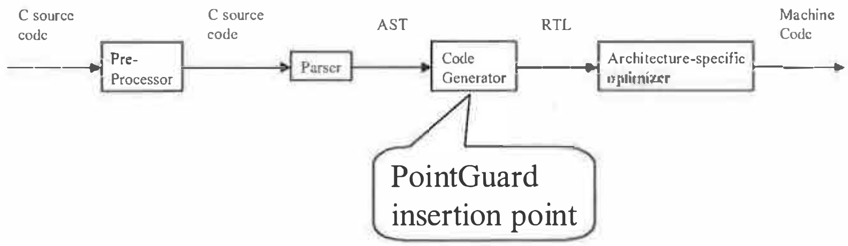


Figure 9 PointGuard in GCC

3.3.2 Actual PointGuard Implementation

Among the three implementation alternatives, we chose to implement PointGuard in the intermediate representation (AST in GCC) as shown in Figure 9. We wanted to implement as *late* as possible in the compiler, to maximize confidence that the PointGuard defenses are not optimized away. AST is the last stage in the GCC compiler where type information is clearly visible, making it possible to distinguish between pointers and other data.

One might be tempted at this point to consider PointGuard-style encryption on *all* loads and stores between memory and CPU registers. However, this defeats the purpose of PointGuard: both legitimate loads and stores and buffer overflows alike would be passed through the PointGuard encryption/decryption, and the buffer overflow attacks would start working again. It is similar to running rot13 encryption *twice*.

The PointGuard encryption method is very basic. The key is a machine-word size (e.g. 32 bits on x86) word, initialized from a suitable source of random keys (in our Linux implementation, `/dev/urandom`) one per process address space. Pointers are encrypted by XORing against the key, and decrypted by XORing again against the same key. This method is simple & efficient; in many cases the single cycle XOR operation will fit neatly into a memory load delay slot (whether implicit or explicit).

This method is cryptographically secure, because cracking it requires guessing the random 32-bit value. Brute force guessing is impractical, because wrong guesses force the victim process to exit, and the new process will have a different key. Massive numbers of processes dying and re-starting is sufficiently "noisy" that conventional intrusion detection methods can detect the attempt at brute force.

Nor can the key be extracted by looking at ciphertext, because the ciphertext is never actually shared with anyone. To obtain a sample of ciphertext, the attacker would have to coerce the victim program into exposing internal pointer values to the attacker. Attackers seeking the privileges of the victim process normally do not have read access to the processes address space. Programs do not normally dump data structures containing pointers outside of their address space, because such pointers lose any meaning outside of the address space.

Thus we cannot identify any feasible means by which the attacker can obtain the PointGuard key.

3.4 PointGuard Special Issues

Apart from the basic problem of building a compiler that correctly emits code implementing the PointGuard defense, Pointguard raises the following special issues. Statically initialized data is described in Section 3.4.1. Protecting the PointGuard key is described in Section 3.4.2. Preventing leaks of clear text pointers is described in Section 3.4.3. Mixed mode code requires programmer intervention, described in Section 3.4.4

3.4.1 Statically Initialized Data

The C language includes support for static initialization of data, including pointers. Normally, these static values are computed at compile time and initialized as the program loads. However, because PointGuard chooses the encryption key at `exec` time, statically initialized pointers cannot be properly initialized until after the program has begun running.

The solution to this is straightforward: we modify the initialization code emitted by the compiler (stuff that runs before `main()`) to re-initialize statically initialized pointers with values encrypted with the current process's key.

3.4.2 Securely Initializing and Protecting the PointGuard Key

Section 3.3.2 described how the attacker cannot feasibly obtain the PointGuard key. But it is also necessary that the attacker not be able to *set* the PointGuard key to a chosen value.

To defend against such an attack, the PointGuard key is stored on its own page when initialized. That page is then made read-only (using mprotect) so that the attacker cannot subsequently use a buffer overflow to change the key value. To make the key page writable again, the attacker would have to execute malicious code, which requires them to bypass PointGuard protection.

3.4.3 Preventing Clear Text Leaks

PointGuard being implemented at the AST level (see Section 3.3.1) the actual CPU registers are not visible. Thus it is possible for the compiler to emit *register spill* instructions that store register contents to the stack in clear text form to make registers available for other purposes. Such register spills are a security threat to PointGuard, because an attacker could potentially use a buffer overflow to corrupt a pointer value stored in clear text form from a register spill, which is subsequently dereferenced by the program when the register values are restored.

To defend against this, a future implementation of PointGuard will *flag* the AST expressions containing PointGuard-relevant expressions such that the flag marks are passed through to the RTL layer. The RTL layer can then notices that PointGuard values are about to be spilled to memory, and insert PointGuard encryption/decryption instructions along with the register spill & restore instructions.

3.4.4 Mixed-mode Code

Because pointers are encrypted, mixed mode code (some PointGuard code, some not) are not compatible unless there is an interface that is aware of the difference, and performs appropriate encryption and decryption. Ideally, this situation should be minimized by compiling all code with PointGuard, as is possible when building an all open source Linux distribution, or in embedded systems.

But in practice, binary-only applications and libraries are sometimes necessary, and so this issue must be handled. There are two cases that need to be handled: PG code calling non-PG code, and non-PG code calling PG code. Both cases are handled with enhancements to function declarations.

```
__std_ptr_mode_on__
#include <stdio.h>
__std_ptr_mode_off__

main() {
      printf("Hello, world\n");
}
```

Figure 10 `hello.c` calling non-PointGuard library functions

To handle PG code calling non-PG code, we specially mark all external function prototype declarations for non-PG code. This is made convenient by the compiler directives `__std_ptr_mode_on__` and `__std_ptr_mode_off__`. Functions declared *between* these directives are marked as needing special marshalling to decrypt arguments before the functions are called. Thus the "hello world" program shown in Figure 10 that is calling out to a non-PG stdio library can make its call to the `printf` library function.

These declarations can be used at finer granularity as storage classes, which is useful for declaring arguments being passed in from non-PG code to PG code. For instance, declaring "`char * __std_ptr_mode_on__ x`" says that x is a *non-encrypted* pointer, and PointGuard will then omit the decryption code when reading this argument. A shorthand notation for this is "`char @x`".

Because the conversion will be handled by the function itself, it does not matter what kind of function prototype the calling code presumed. So for instance, the `printenv` program shown in Figure 11 declares that its second argument is of type "`char @@`" rather than of type "`char **`". The @ in place of * denotes a pointer that is *not* encrypted, and thus needs to be handled specially.

It should be further noted that the @ type declarations need to be carried deeper into the PG code, to the extent that the data structure being passed in has depth. Abstractly, if the non-PG code passes in a pointer to pointer to integer, and the PG code wants to dereference the first pointer and then pass pointer to integer to a second internal PG function, then the internal function needs to also understand that the argument is not encrypted. Concretely, the example shown in Figure 11 handles the `envp` argument, which is of type pointer to pointer to character array, which passes the `envp` to an internal function `print_env()` expecting a similar

```
__std_ptr_mode_on__
#include <stdio.h>
__std_ptr_mode_off__

void
print_env (char @*envp)
{
  int i;

  for (i = 0; envp[i] != NULL; i++)
        printf ("%s\n", envp[i]);
}

int main (int argc, char @@ argv, char @@ env)
{
  print_env (env);
  return 0;
}
```

Figure 11 `printenv.c` being called by non-PointGuard code

pointer to pointer to a character array, which then has to expect an argument of type `char @*`. to accommodate the fact that `main()` has decrypted the outer pointer, but not the inner pointer.

There are also directives `__hashed_ptr_mode_on__` and `__hashed_ptr_mode_off__`. These directives nest within the `__std_ptr_mode_on__` and `__std_ptr_mode_off__` directives (and vice versa) so that, for instance, within a large header file declared standard, selected data structures can be declared to be hashed.

Our current implementation supports this syntax, but specially exempts `varargs` pointer arguments and does not encrypt them. Future implementations will encrypt `varargs` pointer arguments as well, by carrying the pointer's mode (encrypted or not) as part of the function's type signature within the compiler.

Finally, it should be reiterated that the PointGuard specific syntax is necessary *only* to support mixed mode code. Where *all* code is to be compiled with PointGuard, no syntax changes should be required.

4 Compatibility Testing

PointGuard is basically functional, and can compile & run fairly elaborate applications. Unfortunately, for the reasons explained in Section 3.4.4, code needs to be modified to support interoperation with non-PointGuard code. There are two ways to approach this:

Modify the Application: One could modify the application to use `__std_ptr_mode_on__` when ever it makes a call to a library function or kernel system call. This would be a lot of work, and have to be repeated for each application.

Modify the System Libraries: A more cost-effective approach is to modify the system libraries. Libraries are the middleware for applications to interact with the kernel. As such, they are the natural place to insert PointGuard wrappers to encrypt and decrypt pointer data.

Unfortunately, thorough wrapping of system libraries is a lot of work, and we are not yet done. Similar problems with StackGuard compiling system libraries resulted in approximately 6 months delay between a compiler that could compile applications [9] and a compiler that could compile entire systems [7].

5 Security Testing

Our first security test is against the straw man program shown in Figure 12. For illustration, this program places a function pointer (`funcptr`) directly adjacent to an overflowable buffer (`buf`) and then accepts user input that can overflow the buffer and corrupt the function pointer. Exploits for this program without PointGuard are trivial to construct [5].

When the same exploits are tested against a version protected with PointGuard, the victim program crashes with a segmentation fault when it tries to call

```
__std_ptr_mode_on__
   #include <stdio.h>
   #include <stdlib.h>
   #include <unistd.h>
   #include <string.h>
__std_ptr_mode_off__

   #define ERROR -1
   #define BUFSIZE 64

   int goodfunc(const char *str); /* funcptr starts out as this */

   int main(int argc, char @@argv)
   {
      static char buf[BUFSIZE];
      static int (*funcptr)(const char *str);

      if (argc <= 2)
      {
         fprintf(stderr, "Usage: %s <buf> <goodfunc arg>\n", argv[0]);
         exit(ERROR);
      }

      setuid(0);
      funcptr = (int (*)(const char *str))goodfunc;
      memset(buf, 0, sizeof(buf));
      strncpy(buf, argv[1], strlen(argv[1]));
      (void)(*funcptr)(argv[2]);
      return 0;
   }
```

Figure 12 Straw man vulnerability overflowing an adjacent function pointer

`funcptr`. To ensure that the crash is not due to simple variation in program layout, we constructed a *special* version of PointGuard that uses a value of 0 for the encryption key (which has no encryption effect under XOR) and created exploits that actually work against the null-key PointGuard program. These same exploits again produce repeated segmentation faults against the victim program protected with PointGuard.

There are ample live examples of heap and static data overflows that PointGuard is intended to block, such as telnetd [26], WU-FTPd [27], CVS [15], and sudo [20]. Unfortunately, these exploits almost all attack `malloc` data structures, which are part of `glibc`. Until we have a PointGuard version of `glibc`, effective security testing of these programs against live exploits is problematic.

That PointGuard might stop real exploits from penetrating real vulnerabilities in real programs would be comforting, but does not assure that PointGuard is non-bypassable. This is important, because simple obscurity tricks are sufficient to cause many exploits to fail, but these exploits would soon succeed if simply re-tuned to bypass the obscurity defenses.

Unfortunately, it is not possible to use testing and experimentation to show non-bypassability: testing can only show bypassability. Non-bypassability must be established by inspection. Our argument is that:

1. Bypassing PointGuard is defined as hijacking a program by corrupting one or more pointers.
2. *Usefully* corrupting a pointer requires pointing it at a *specific* location.
3. Under PointGuard protection, a pointer cannot be corrupted to point to a specific location without knowing the secret key.
4. Learning the secret key requires either obtaining the secret key directly, or cryptanalysis against a sample pointer value.

5. Obtaining the secret key directly would require corrupting a pointer precisely, which begs the question (see Section 3.4.2).
6. Obtaining a sample of ciphertext (an encrypted pointer) would require either corrupting a pointer precisely (which begs the question) or a program that leaks pointer values (which is highly unusual).

Thus it should be difficult to bypass PointGuard and control a program by corrupting a pointer. However, it is possible for attackers to "bypass" PointGuard by attacking non-pointer objects, such as overflowing one character array to change the string value of an adjacent character array. Such an attack could, e.g. cause a program to change which program the victim program is about to `exec()`. This form of attack is out of scope for PointGuard's protection.

6 Performance Testing

We measure the overhead imposed by PointGuard using microbenchmarks described in Section 6.1 and macrobenchmarks described in Section 6.2. All benchmarks were run on a 1.6 GHz P4-M with 512 KB of L2 cache and 512 MB of DRAM, with the compiler set to schedule for i686:

6.1 Microbenchmarks

The microbenchmarks comprised three tests:

Read: exercise reading pointers by following a linked list.

Write: stores values into an array of pointers.

Read/Write: copies values from one array of pointers to another array of pointers.

Tests were also partitioned into *cachable* and *large*. The cachable tests fit in the L2 cache, while the large tests did not. The rationale being that the cachable case presents smaller/fewer load delay slots (the load delay slot being where PointGuard decryption happens) and thus should expose greater PointGuard overhead.

The cachable microbenchmark results are shown in Table 1, and non-cachable in Table 2. Counter-intuitively, PointGuard imposed less overhead on the cachable case than on the non-cachable case, and in fact PointGuard provided substantial performance *improvements* vs. the standard GCC in many cases.

We believe these performance improvements to be the result of PointGuard imposing heavier use of registers for dereferencing pointers. We do not claim that PointGuard itself is actually enhancing performance, but rather that the PointGuard implementation has exposed the possibility that GCC could provide greater performance if it made more aggressive use of registers.

Table 1: Microbenchmark Results for Cachable

	Non-Optimized	-O2	-O3, -fomit-frame-ptrs
Read	-1.47%	0.94%	-0.79%
Write	2.78%	-11.45%	-11%
RW	-1.01%	-9.93%	-9.83%

Table 2: Microbenchmark Results for Non-cachable

	Non-Optimized	-O2	-O3, -fomit-frame-ptrs
Read	-0.34%	-0.14%	-0.08%
Write	4.02%	3.46%	2.44%
RW	-0.85%	-1.45%	-0.75%

Table 3: SciMark Results for Cachable

	Non-Optimized	Optimized
FFT	-2.12%	-6.09%
SOR	0.20%	-0.21%
Monte	0.63%	0.33%
Sparse	0.26%	-0.16%
LU	-2.95%	-0.76%
Composite	-0.95%	-1.13%

6.2 Macrobenchmarks

Our first macrobenchmark is the SciMark benchmark suite [29], a compute-intensive benchmark for scientific computing. Again, this benchmark was partitioned into cachable and non-cachable data sets.

Our cachable SciMark results are shown in Table 3, and non-cachable results in Table 4. Results are similar to the microbenchmarks, but the performance gains if any are smaller, and performance losses are higher.

Table 4: SciMark Results for Non-cachable

	Non-Optimized	Optimized
FFT	5.43%	4.16%
SOR	-0.06%	0.06%
Monte	0.28%	0.06%
Sparse	0.00%	-0.07%
LU	-0.22%	-0.63%
Composite	0.08%	-0.05%

Table 5: OpenSSL Speed Throughput

Cipher	Normal	PG	% Overhead
MD5 8KB	86,794	88,989	-2.5%
SHA1 8KB	56,180	58,119	-3.5%
DES CBC 8KB	10,619	9336	12%
DES EDE3 8KB	3902	3639	6.7%
Blowfish CBC 8KB	49,987	39,316	21%
RSA 1Kbits	1094	892	18%
DSA 2Kbits	23	19	17%

Our second macrobenchmark is the OpenSSL Speed, a part of the OpenSSL package [11]. This is a benchmark included with the SSL package to measure throughput on various ciphers, in terms of how many units of work can be done in a 10 second period.

Our results for OpenSSL Speed are shown in Table 5. As in the microbenchmarks, performance varies from a nominal speedup of 3.5% to a slowdown of 21%. The variation is due to instruction scheduling and register usage: PointGuard increases register pressure, and consumes load delay slots in CPU scheduling. Increasing register pressure improves performance where registers are available, and decreases performance if registers were already fully occupied. Increased use of delay slots is free if there were empty delay slots, and induces overhead if delay slots were already full.

7 Related Work

Previously we surveyed buffer overflow attacks and defenses [10]. Attacks were classified according to how the malicious code was injected, and how the victim program is coerced to jump to the malicious code. Correspondingly, defenses were classified according to how they stopped these effects.

The related work presented here is somewhat updated, including results produced since our previous survey. A summary is shown in Table 6. "Class" indicates which of the families of technologies a defense can be grouped with. "Coverage" indicates the classes of threats that the technology addresses. "Bypassable" indicates whether the attacker can, with some effort, craft an attack that will bypass the defense and exploit a vulnerability anyway. "Cost" indicates the cost in either performance or software developer effort.

Note that many of these technologies have distinct areas of coverage, and can be combined to achieve greater coverage. Section 7.1 describes Bounds Checking. Section 7.2 describes various non-executable buffers. Section 7.3 describes address space randomization. Section 7.4 describes pointer protection.

7.1 Bounds Checking

Bounds checking provides "perfect" protection against buffer overflows, but at a substantial cost in compatibility, performance, or both.

Jones & Kelly [19] and Brugge [31] provide full bounds checking for C code while maintaining `sizeof(void *) == sizeof(int)`, which is important for preserving code compatibility with legacy systems. This compiler does so by using associative lookup on each pointer reference to an array descriptor that stores base and bounds. Performance penalties are high, approximately 10X to 30X slowdown.

The Bounded Pointers project [22] also provides full bounds checking, but changes pointers from a single word into a tuple that incorporates base and bounds. This improves performance by eliminating the associative lookup in Jones&Kelly, but costs compatibility because pointers no longer fit in a single word. Performance penalties are still high at approximately 5X slowdown.

More general than bounds checking, *type safety* subsumes bounds checking and protects against all manner of data type chicanery, not just buffer overflows. Classically, strongly typed languages such as Java and ML provide strong static type safety, built in to the program-

ming language semantics. Because the programming languages were designed for type safety, efficiency is high, and compatibility is not an issue.

More recently, hybrid languages designed to be *safer* versions of the C programming language such as CCured [24] and Cyclone [18] have appeared. These are *dialects* of C, removing some unsafe constructs, and adding others. This provides a safe programming environment with good performance, with stronger security than PointGuard, but also imposes a different order of magnitude on the developer to achieve that safety. A developer can port an application to these safer dialects in a few hours or days, where as PointGuard was designed to allow a developer to compile & protect *millions* of lines of code in a few hours or day.

7.2 Non-Executable Buffers

Making various pieces of memory non-executable restricts where attack payload can be injected. This is good because it is fast, transparent, and works on binary-only applications. The main limitation is that this defense can be bypassed, because suitable attack payload code (effectively "`exec(sh)`")) is almost always resident in victim program address spaces, and so pointer corruption is all that is necessary for the determined attacker to succeed.

Non-executable stack segments [12, 14] was one of the first general purpose defenses against buffer overflows. Zero performance cost and near-zero compatibility cost, but can be bypassed.

The PAX project provides non-executable heap, making it more difficult to bypass, but still can be bypassed. Performance cost is substantial at 10% overall.

7.3 Address Space Randomization

PAX also incorporates ASLR (Address Space Layout Randomization) which can be viewed as the dual of PointGuard: rather than randomizing pointers, ASLR randomizes the location of key memory objects. Benefits are similar to PointGuard, but because objects that are randomly located are coarser, there is residual risk of attackers exploiting adjacency and approximate memory location.

Sekar et al [3] have a new implementation of this concept that randomizes more elements of the address space

Table 6: Buffer Overflow Defenses

Class	Technology	Coverage	Bypassable	Cost
Bounds Checking	Jones&Kelly	complete	no	10X to 30X
	Bounded Pointers	complete	no	3-5X
	Safe Languages	complete	no	complete rewrite
	Safer C Dialects	complete	no	port software
Non-executable Buffers	Non-executable stack	stack buffers	yes	0
	Non-executable heap	heap buffers	yes	10-30%
Address Space Randomization	PAX/ASLR	buffer overflows that don't depend on adjacency	probably	~0
	Sekar et al	buffer overflows that don't depend on adjacency	maybe	0-18%
Pointer Protection	StackGuard	activation records	no	~0
	Libsafe	library string functions attacking activation records	yes	~0
	PointGuard	pointers	maybe	0-20%

layout, which may make it harder to bypass than PAX/ASLR.

7.4 Pointer Protection

Libsafe [2] provides plausibility checks on the arguments to the "big 7" string manipulation functions in the standard C library. Libsafe imposes low overhead, and compatibility is excellent, providing protection to binary-only applications. Protection, however, is limited only to vulnerabilities involving the protected 7 functions.

Snarskii [30] introduced pointer integrity checking with a `libc` library for FreeBSD that checked the integrity of activation records created within the library. StackGuard generalized this technique from a single protected library into a compiler, at first using the Random and Terminator Canaries [8]. In 1999 we introduced the XOR canary to address Emsi's Attack [4]. The XOR canary helped improve the PointGuard design by changing PointGuard from using adjacent canaries to directly encrypting pointers. StackGhost [16] provides a similar degree of protection to StackGuard, using the SPARC CPU's register spill detection hardware.

8 Conclusions

PointGuard provides protection against all vulnerabilities related to pointer corruption, which includes most current and anticipated buffer overflow, as well as related attacks such as printf format bugs and multiple free errors. PointGuard imposes minimal performance overhead, compatibility and performance overhead.

9 Availability

When PointGuard is complete, it will be released under the terms of the GPL from `http://immunix.com/`

10 Acknowledgments

We would like to thank the developers of the open source GCC compiler who made this work possible, DARPA for funding this work, Chris Wright and Seth Arnold for hard labor helping to debug, and Jane-Ellen Long of USENIX for help in final preparation of the paper.

References

[1] Anonymous. Once upon a free()... *Phrack*, 11(57), August 2001.

[2] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent Run-Time Defense Against Stack Smashing Attacks. In *2000 USENIX Annual Technical Conference*, San Diego, CA, June 18-23 2000.

[3] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address Obfuscation: An Approach to Combat Buffer Overflows, Format-String Attacks, and More. In *12th USENIX Security Symposium*, Washington, DC, August 2003.

[4] "Bulba" and "Kil3r". Bypassing stackguard and stackshield. *Phrack*, 10(56), May 2000.

[5] Matt Conover. w00w00 on Heap Overflows. http://www.w00w00.org/files/articles/heaptut.txt, 1999.

[6] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. FormatGuard: Automatic Protection From printf Format String Vulnerabilities. In *USENIX Security Symposium*, Washington, DC, August 2001.

[7] Crispin Cowan, Steve Beattie, RyanFinnin Day, Calton Pu, Perry Wagle, and Erik Walthinsen. Protecting Systems from Stack Smashing Attacks with StackGuard. In *Linux Expo*, Raleigh, NC, May 1999.

[8] Crispin Cowan, Tim Chen, Calton Pu, and Perry Wagle. StackGuard 1.1: Stack Smashing Protection for Shared Libraries. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 1998. Brief presentation and poster session.

[9] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *7th USENIX Security Conference*, pages 63–77, San Antonio, TX, January 1998.

[10] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. In *DARPA Information Survivability Conference and Expo (DISCEX)*, January 2000. Also presented as an invited talk at SANS 2000, March 23-26, 2000, Orlando, FL, http://schafercorp-ballston.com/discex}.

[11] Mark J. Cox, Ralf S. Engelschall, Stephen Henson, and Ben Laurie. openSSL. http://www.openssl.org/, March 2001.

[12] "Solar Designer". Non-Executable User Stack.

http://www.openwall.com/linux/.

[13] "Solar Designer". JPEG COM Marker Processing Vulnerability in Netscape Browsers. http://online.securityfocus.com/archive/1/71598, July 25 2000. Bugtraq.

[14] Casper Dik. Non-Executable Stack for Solaris. Posting to `comp.security.unix`, http://x10.dejanews.com/getdoc.xp?AN=207344316&CONTEXT=890082637.156735%9211&hitnum=69&AH=1}, January 2 1997.

[15] Igor Dobrovitski. Exploit for CVS double free() for Linux pserver. http://online.securityfocus.com/archive/1/309913, February 2 2003. Bugtraq.

[16] Mike Frantzen and Mike Shuey. StackGhost: Hardware Facilitated Stack Protection. In *USENIX Security Symposium*, Washington, DC, August 2001.

[17] Intel. *IA-32 Intel Architecture Software Developer's Manual*. Intel Corporation, 2002.

[18] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A Safe Dialect of C. In *Proceedings of USENIX Annual Technical Conference*, Monteray, CA, June 2002.

[19] Richard Jones and Paul Kelly. Bounds Checking for C. http://www-ala.doc.ic.ac.uk/ phjk/BoundsChecking.html, July 1995.

[20] Michel Kaempf. [synnergy] - Sudo Vudo. http://online.securityfocus.com/archive/1/189037, June 6 2001. Bugtraq.

[21] BrianW. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, second edition, 1988.

[22] Greg McGary. Bounds Checking in C & C++ Using Bounded Pointers. http://gcc.gnu.org/projects/bp/main.html, 2000.

[23] Motorola, Inc. *MC68020 32-bit Microprocessor User's Manual*, second edition, 1984.

[24] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL02)*, London, England, January 2002. Also available at http://raw.cs.berkeley.edu/Papers/ccured_popl02.pdf.

[25] "Aleph One". Smashing The Stack For Fun And Profit. *Phrack*, 7(49), November 1996.

[26] "Zen Parse". ADV/EXP: netkit less than 0.17 in.telnetd remote buffer overflow. http://online.securityfocus.com/archive/1/203000, August 10 2001. Bugtraq.

[27] "Zen Parse". Re: exploiting wu-ftpd. http://online.securityfocus.com/archive/82/244938, December 12 2001. Bugtraq.

[28] DavidA. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, first edition, 1990.

[29] Roldan Pozo and Bruce Miller. SciMark 2.0. http://math.nist.gov/scimark, June 20 2000.

[30] Alexander Snarskii. FreeBSD Stack Integrity Patch. ftp://ftp.lucky.net/pub/unix/local/libc-letter, 1997.

[31] Herman ten Brugge. Bounds Checking C Compiler. http://web.inter.NL.net/hcc/Haj.Ten.Brugge/, 1998.

Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits

Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar
Department of Computer Science,
Stony Brook University, Stony Brook, NY 11794
{sbhatkar,dand,sekar}@cs.sunysb.edu

Abstract

Attacks which exploit memory programming errors (such as buffer overflows) are one of today's most serious security threats. These attacks require an attacker to have an in-depth understanding of the internal details of a victim program, including the locations of critical data and/or code. *Program obfuscation* is a general technique for securing programs by making it difficult for attackers to acquire such a detailed understanding. This paper develops a systematic study of a particular kind of obfuscation called *address obfuscation* that randomizes the location of victim program data and code. We discuss different implementation strategies to randomize the absolute locations of data and code, as well as relative distances between data locations. We then present our implementation that transforms object files and executables at link-time and load-time. It requires no changes to the OS kernel or compilers, and can be applied to individual applications without affecting the rest of the system. It can be implemented with low runtime overheads. Address obfuscation can reduce the probability of successful attacks to be as low as a small fraction of a percent for most memory-error related attacks. Moreover, the randomization ensures that an attack that succeeds against one victim will likely not succeed against another victim, or even for a second time against the same victim. Each failed attempt will typically crash the victim program, thereby making it easy to detect attack attempts. These aspects make it particularly effective against large-scale attacks such as Code Red, since each infection attempt requires significantly more resources, thereby slowing down the propagation rate of such attacks.

1 Introduction

The C and C++ languages are popular primarily because of the precise low-level control they provide over system resources, including memory. Unfortunately, this control is more than most programmers can handle, as evidenced by the host of memory-related programming errors which plague software written in these languages, and continue to be discovered every day. Attacks which exploit memory errors such as buffer overflows constitute the largest class of attacks reported by organizations such as the CERT Coordination Center, and pose a serious threat to the computing infrastructure.

To date, a number of attacks which exploit memory errors have been developed. The earliest of these to achieve widespread popularity was the *stack smashing* attack [31, 27], in which a stack-allocated buffer is intentionally overflowed so that a return address stored on the stack is overwritten with the address of injected malicious code. (See Figure 1). To thwart such attacks, several approaches were developed, which, in one way or another, prevent undetected modifications to a function's return address. They include the StackGuard [11] approach of putting *canary values* around the return address, so that stack smashing can be detected when the canary value is clobbered; saving a second copy of return address elsewhere [9, 6]; and others [16].

The difficulty with the above approaches is that while they are effective against stack-smashing attacks, they can be defeated by attacks that modify code pointers in the static or heap area. In addition, attacks where control flow is not changed, but security-critical data such as an argument to `chmod` or `execve` system call are changed, are not addressed. Recently, several new classes of vulnerabilities such as the integer overflow vulnerability (reported in Snort [34] and earlier in sshd [35]), heap overflows [23] and double-free vulnerabilities [2] have emerged. These developments lead us to conclude that additional ways to exploit the lack of memory safety in C/C++ programs will continue to be discovered in the future. Thus, it is important to develop approaches that provide systematic protection against all foreseeable memory error exploitations.

As a first step towards developing more comprehensive solutions against memory exploits, we observe that such exploits require an attacker to possess a detailed understanding of the victim program, and have precise knowl-

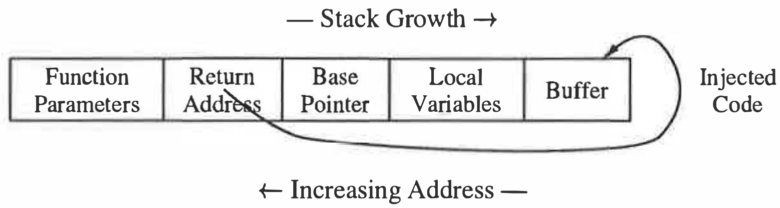


Figure 1: A buffer overflow in which the current function's return address is replaced with a pointer to injected code.

edge of the organization of data and code within the victim program memory. *Code obfuscation* is a general technique that attempts to secure programs by making them hard to understand. It is typically implemented using a set of randomized, semantics-preserving program transformations [38, 10, 4]. While code obfuscation is concerned primarily with preventing the understanding and reverse engineering of binary code, our interest lies in obfuscations which modify the internal runtime behavior of programs in ways that don't affect the observable semantics, but do create unpredictability which makes it difficult to successfully craft attacks which exploit memory errors.

Forrest, et.al. [17] suggested the use of randomized program transformations as a way to introduce *diversity* into applications. Such diversity makes it necessary for attackers to analyze each copy of the victim program independently, thereby greatly increasing the cost of developing attacks. They presented a prototype implementation that performed one particular kind of randomization: the randomization of the addresses of stack-resident data. Their implementation modified the gcc compiler to insert a random amount of padding into each stack frame. Our paper extends this basic idea, and presents a systematic study of the range of address randomizations that can be achieved using program transformation.

Address obfuscation is a program transformation technique in which a program's code is modified so that each time the transformed code is executed, the virtual addresses of the code and data of the program are randomized. As we will show, this makes the effect of most memory-error exploits non-deterministic, with only a very small chance of success. Attackers are forced to make many attempts on average before an attack succeeds, with each unsuccessful attack causing the target program to crash, increasing the likelihood that the attack will be detected. Moreover, an attack that succeeds against one victim will not succeed against another victim, or even for a second time against the same victim. This aspect makes it particularly effective against large-scale attacks such as Code Red, since each infection attempt requires significantly more resources, thereby greatly reducing the propagation rate of such attacks.

The PaX project has also developed an approach for randomizing the memory regions occupied by program code and data, called Address Space Layout Randomization (ASLR) (See `http://pageexec.virtualave.net` for documentation on PaX project.) Rather than viewing address obfuscation as a program transformation, they view it as an operating system feature. In particular, they have modified the Linux kernel so that it randomizes the base address of different sections of memory, such as the stack, heap, code, and memory-mapped segments. A key benefit of this approach is that it requires no changes to individual applications (other than having the compiler generate position-independent code). However, since the approach incorporates no analysis of the applications, it is difficult to perform address randomizations beyond changes to the base addresses of different memory segments. In contrast, a program transformation approach will permit randomization of the locations of individual variables and routines within these memory sections. Such randomization makes it difficult to carry out attacks that rely on relative distances between variables to modify critical data, e.g., a string used as an argument to `execve`. Moreover, it introduces significant additional diversity into the program, as it is no longer possible to craft attacks by knowing just the offsets in the base address of various memory segments. (These offsets can potentially be learned by exploiting vulnerabilities that may allow attackers to read contents of victim program memory without crashing it.)

The current generation of compilers and application binary interfaces limit how much randomization is possible, and at what stage (compile-time, link-time, load-time or runtime) such randomization can be performed. Our implementation focuses on techniques that can be smoothly integrated into existing OS environments. The key contribution of this paper is to develop and analyze the range of address obfuscations that can be implemented effectively with low runtime overheads. The principal benefits of this approach are:

- It systematically protects against a wide range of at-

Technique	Attack Target									
	Code Pointer					Data				
	Return Address	Frame Pointer	Function Pointer		Dynamic Linker Tables	Pointer		Non-Pointer		
			Stack	Static/Heap		Stack	Static/Heap	Stack	Static	Heap
StackGuard [11]	√*									
Libverify [6], RAD [9]	√††									
Etoh and Yoda [16]	√*	√*	√*			√*		√*		
PointGuard [13]	√‡	√	√	√	√	√	√			
Address Obfuscation	√‡	√	√	√	√	√	√	√†	√**	√†

* Only protected from buffer-overflow attacks; no protection from other attacks.

† Limited protection provided.

** Possible in principle but not currently implemented.

†† Susceptible to attacks that simultaneously corrupt return address and another location (second copy of return address)

‡ Some susceptibility to attacks that corrupt return address and another stack-resident pointer.

Figure 2: Targets of memory error exploits, and effectiveness of defenses against them.

tacks which exploit memory programming errors, including stack smashing, heap-overflow, integer overflow, and typical format-string attacks.

- It can be easily applied to existing legacy code without modifying the source code, or the underlying operating system. Moreover, it can be applied selectively to protect security-critical applications without needing to change the rest of the system.
- The transformation is fast and introduces only a low runtime overhead.

Applicability to legacy code without source-code or operating system changes provides an easy migration path for existing systems to adopt address obfuscation. Such a solution can also be ported more easily to proprietary operating systems. Finally, the approach can be easily combined with existing techniques, such as Stackguard and Formatguard, to provide additional security.

1.1 Overview of Address Obfuscation and How it Works.

We start with the observation that the goal of an attacker is to cause the target program to execute attack-effecting code. This code itself may be provided by the attacker (*injected code*), or it may already be a part of the program (*existing code*). A direct way to force execution of such code is through a change to the control flow of the program. This requires the attacker to change a code pointer stored somewhere in memory, so that it points to the code of their choice. In such a case, when the corrupted code pointer is used as the target of a jump or call instruction, the program ends up executing the code chosen by the attacker. Some natural choices for such code pointers include the return address (stored on the stack), function pointers (stored on the stack, static area or the heap), the global offset table (GOT) that is used in the context of dynamic linking, and buffers storing `longjmp` data. An indirect way to force execution of attack-effecting code is to change security-critical data that is used by the program in its normal course of execution. Examples of such data include arguments to a `chmod` or `execve` system call, variables holding security critical data such as a flag indicating whether a user has successfully authenticated herself, etc.

There are essentially two means by which an attacker can exploit a memory error: by overwriting a pointer value, or by overwriting non-pointer data. Since code sections cannot be overwritten in most modern operating systems, there are three possible combinations of goals and means: corrupting a code-pointer, corrupting a data-pointer, or corrupting non-pointer data. Of these, the two pointer-corrupting attacks involve overwriting a pointer with the address of data or code chosen by the attacker. These two kinds of attacks require the attacker to know the absolute address of such data or code, and hence we call them *absolute address-dependent* attacks. The third kind of attack is called *relative address-dependent*, because it does not overwrite pointers, and requires only relative address information — in particular, an attacker needs to know the relative distance between a buffer (which is overrun) and the location of the data item to be corrupted. Figure 2 shows these three classes of attacks, further subdivided based on the pointer or data value that is targeted by an attack. It shows which of today's protection schemes (including ours) protect against them. As it shows, Stackguard, Libverify and RAD protect against buffer overrun attacks that overwrite the return address. PointGuard [13] is an approach that encrypts stored pointer values (by xor-ing them with a random number). It can be thought of as obfuscating pointer values as opposed to the addresses pointed by them. The benefit of their approach is that the probability of an attack making a successful guess is smaller than with ad-

dress obfuscation. A drawback is that it does not provide protection against attacks that modify non-pointer values, e.g., attacks that modify critical data, or integer subscripts. A concrete example of such an attack is the recent integer overflow exploit [20], which is protected by address obfuscation. The PaX project's ASLR approach provides protection against pointer-based attacks in much the same way as address obfuscation, but not against data attacks that exploit relative distances between variables. A more detailed comparison of our approach with these approaches can be found in Sections 5 and 3.

1.2 Organization of the Paper.

The rest of this paper is organized as follows. In Section 2, we describe several possible obfuscating transformations, and describe our implementation approach. Section 3 discusses the effectiveness of our approach against different attacks, and analyzes the probability of mounting successful attacks. Runtime overheads introduced by our approach are discussed in Section 4, followed by a discussion of related work in Section 5. Finally, Section 6 provides a summary and discusses future work.

2 Address Obfuscation

2.1 Obfuscating Transformations

The objectives of address obfuscation are to (a) randomize the absolute locations of all code and data, and (b) randomize the relative distances between different data items. These objectives can be achieved using a combination of the following transformations:

I. Randomize the base addresses of memory regions.

By changing the base addresses of code and data segments by a random amount, we can alter the absolute locations of data resident in each segment. If the randomization is over a large range, say, between 1 and 100 million, the virtual addresses of code and data objects become highly unpredictable. Note that this does not increase the physical memory requirements; the only cost is that some of the virtual address space becomes unusable. The details depend on the particular segment:

1. *Randomize the base address of the stack.* This transformation has the effect of randomizing all the addresses on the stack. A classical stack-smashing attack requires the return address on the stack to be set to point to the beginning of a stack-resident buffer into which the attacker has injected his/her code. This becomes very difficult when the attacker cannot predict the address of such a buffer due to randomization of stack addresses. Stack-address randomization can be implemented by subtracting a large random value from the stack pointer at the beginning of the program execution.
2. *Randomize the base address of the heap.* This transformation randomizes the absolute locations of data in the heap, and can be performed by allocating a large block of random size from the heap. It is useful against attacks where attack code is injected into the heap in the first step, and then a subsequent buffer overflow is used to modify the return address to point to this heap address. While the locations of heap-allocated data may be harder to predict in long-running programs, many server programs begin execution in response to a client connecting to them, and in this case the heap addresses can become predictable. By randomizing the base address of the heap, we can make it difficult for such attacks to succeed.
3. *Randomize the starting address of dynamically-linked libraries.* This transformation has the effect of randomizing the location of all code and static data associated with dynamic libraries. This will prevent *existing code* attacks (also called *return-into-libc* attacks), where the attack causes a control flow transfer to a location within the library that is chosen by the attacker. It will also prevent attacks where static data is corrupted by first corrupting a pointer value. Since the attacker does not know the absolute location of the data that he/she wishes to corrupt, it becomes difficult for him/her to use this strategy.
4. *Randomize the locations of routines and static data in the executable.* This transformation has the effect of randomizing the locations of all functions in the executable, as well as the static data associated with the executable. The effect is similar to that of randomizing the starting addresses of dynamic libraries.

We note that all of the above four transformations are also implemented in the PaX ASLR system, but their implementation relies on kernel patches rather than program transformations. The following two classes of transformations are new to our system. They both have the effect of randomizing the relative distance between the locations of two routines, two variables, or between a variable and a routine. This makes it difficult to develop successful attacks that rely on adjacencies between data items or routines. In addition, it introduces additional randomization into the addresses, so that an attacker that has somehow learned the offsets of the base addresses will still have difficulty in crafting successful attacks.

II. Permute the order of variables/routines.

Attacks that exploit relative distances between objects, such as attacks that overflow past the end of a buffer to

overwrite adjacent data that is subsequently used in a security-critical operation, can be rendered difficult by a random permutation of the order in which the variables appear. Such permutation makes it difficult to predict the distance accurately enough to selectively overwrite security-critical data without corrupting other data that may be critical for continued execution of the program. Similarly, attacks that exploit relative distances between code fragments, such as partial pointer overflow attacks (see Section 3.2.3), can be rendered difficult by permuting the order of routines. There are three possible rearrangement transformations:

1. *permute the order of local variables in a stack frame*
2. *permute the order of static variables*
3. *permute the order of routines in shared libraries or the routines in the executable*

III. Introduce random gaps between objects.

For some objects, it is not possible to rearrange their relative order. For instance, local variables of the caller routine have to appear at addresses higher than that of the callee. Similarly, it is not possible to rearrange the order of malloc-allocated blocks, as these requests arrive in a specific order and have to be satisfied immediately. In such cases, the locations of objects can be randomized further by introducing random gaps between objects. There are several ways to do this:

1. *Introduce random padding into stack frames.* The primary purpose of this transformation is to randomize the distances between variables stored in different stack frames, which makes it difficult to craft attacks that exploit relative distances between stack-resident data. The size of the padding should be relatively small to avoid a significant increase in memory utilization.
2. *Introduce random padding between successive* `malloc` *allocation requests.*
3. *Introduce random padding between variables in the static area.*
4. *Introduce gaps within routines, and add jump instructions to skip over these gaps.*

Our current implementation supports all the above-mentioned transformations for randomizing the base addresses of memory regions, none of the transformations to reorder variables, and the first two of the transformations to introduce random gaps.

2.2 Implementation Issues

There are two basic issues concerning the implementation of the above-mentioned transformations. The first concerns the timing of the transformations: they may be performed at compile-time, link-time, installation-time, or load-time. Generally speaking, higher performance can be obtained by performing transformations closer to compilation time. On the other hand, by delaying transformations, we avoid making changes to system tools such as compilers and linkers, which makes it easier for the approach to be accepted and used. Moreover, performing transformations at a later stage means that the transformations can be applied to proprietary software that is distributed only in binary form.

The second implementation issue is concerned with the time when the randomization amounts are determined. Possible choices here are (a) transformation time, (b) beginning of program execution, and (c) continuously changing during execution. Clearly, choice (c) increases the difficulty of attacks, and is hence preferred from the point of security. Choices (a) or (b) may be necessitated due to performance or application binary interface compatibility considerations. For instance, it is not practical to remap code at different memory locations during program execution, so we cannot do any better than (b) for this case. In a similar manner, adequate performance is difficult to obtain if the relative locations of variables with respect to some base (such as the frame pointer for local variables) is not encoded statically in the program code. Thus, we cannot do any better than choice (a) in this case. However, choice (a) poses some special problems: it allows an attacker to gradually narrow down the possibilities with every attack attempt, since the same code with the same randomizations will be executed after a crash. To overcome this problem, our approach is to periodically re-transform the code. Such retransformation may take place in the background after each execution, or it may take place after the same code is executed several times. With either approach, there still remains one problem: a local attacker with access to such binaries can extract the random values from the binary, and use them to craft a successful attack. This can be mitigated by making such executables unreadable to ordinary users. However, Linux currently makes the memory maps of all processes to be readable (through the special file `/proc/pid/maps`), which means a local user can easily learn the beginning of each memory segment, which makes it much easier to defeat address obfuscation. In particular, the attacker can easily figure out the locations of the code segments, which makes it possible to craft existing code attacks. This is a limitation of our current implementation.

Our approach is to delay the transformation to the latest possible stage where adequate performance is obtainable. In our current implementation, the transformation is performed on object files (i.e., at link-time) and executables. For ease of implementation, we have

fixed many randomizations at transformation time, such as the gaps introduced within the stack frame for any given function, the locations where libraries are loaded, etc. This means that programs have to be periodically (or frequently) re-obfuscated, which may be undesirable as the obfuscation interacts with other security procedures such as integrity-checking of executables. We therefore plan to move towards options (b) and (c) in the future.

Next, we describe our approach for implementing most of the above-mentioned transformations. Our implementation targets the Intel x86 architectures running ELF-format [30] executables on the Linux operating system.

2.3 Implementation Approach

Our implementation transforms programs at the binary level, inserting additional code with the LEEL binary-editing tool [40]. The main complication is that on most architectures, safe rewriting of machine code is not always possible. This is due to the fact that data may be intermixed with code, and there may be indirect jumps and calls. These two factors make it difficult to extract a complete control-flow graph, which is necessary in order to make sure that all code is rewritten as needed, without accidentally modifying any data. Most of our transformations, such as stack base randomization are simple, and need to be performed in just one routine, and hence are not impacted by the difficulty of extracting an accurate control-flow graph. However, stack-frame padding requires a rewrite of all the routines in the program and libraries, which becomes a challenge when some routines cannot be accurately analyzed. We take a conservative approach to overcome this problem, rewriting only those routines that can be completely analyzed. Further details can be found in Section 2.3.4.

2.3.1 Stack base address randomization

The base address of the stack is randomized by extra code which is added to the text segment of the program. The code is spliced into the execution sequence by inserting a jump instruction at the beginning of the main routine. The new code generates a random number between 1 and 10^8, and decrements the stack pointer by this amount. In addition, the memory region corresponding to this "gap" is write-protected using the `mprotect` system call. The write-protection ensures that any buffer overflow attacks that overflow beyond the base of the stack into the read-only region will cause the victim program to crash.

2.3.2 DLL base address randomization

In the ELF binary format, the program header table (PHT) of an executable or a shared library consists of a set of structures which hold information about various segments of a program. Loadable segments are mapped to virtual memory using the addresses stored in the `p_vaddr` fields of the structures (for more details, see [30]). Since executable files typically use (non-relocatable) absolute code, the loadable segments must reside at addresses specified by `p_vaddr` in order to ensure correct execution.

On the other hand, shared object segments contain position-independent code (PIC), which allows them to be mapped to almost any virtual address. However, in our experience, the dynamic linker almost always chooses to map them starting at `p_vaddr`, e.g., this is the case with `libc.so.6` (the Standard C library) on Red Hat Linux distributions. The lowest loadable segment address specified is `0x42000000`. Executables start at virtual address `0x08048000`, which leaves a large amount of space (around 927MB) between the executable code and the space where shared libraries are mapped. Typically, every process which uses the dynamically-linked version of `libc.so.6` will have it mapped to the same base address (`0x42000000`), which makes the entry points of the `libc.so.6` library functions predictable. For example, if we want to know the virtual address where function `system()` is going to be mapped, we can run the following command:

```
$ nm /lib/i686/libc.so.6 | grep system
42049e54 T __libc_system
2105930 T svcerr_systemerr
42049e54 W system
```

The third line of the output shows the virtual address where `system` is mapped.

In order to prevent existing code attacks which jump to library code instead of injected code, the base address of the libraries should be randomized. There are two basic options for doing this, depending on when the randomization occurs. The options are to do the randomization (1) once per process invocation, or (2) statically. The trade-offs involved are as follows:

1. *Dynamically randomize library addresses using* `mmap`. The dynamic linker uses the `mmap` system call to map shared libraries into memory. The dynamic linker can be instrumented to instead call a wrapper function to `mmap`, which first randomizes the load address and then calls the original `mmap`. The advantage of this method is that in every program execution, shared libraries will be mapped to different memory addresses.
2. *Statically randomize library addresses at link-time.* This is done by dynamically linking the executable with a "dummy" shared library. The dummy library need not be large enough to fill the virtual address

space between the segments of the executable and standard libraries. It can simply introduce a very large random gap (sufficient to offset the base addresses of the standard libraries) between the load-addresses of its `text` and `data` segments. Since shared libraries use relative addressing, the segments are mapped along with the gap.

On Linux systems, the link-time gap can be created by using the `ld` options `-Tbss`, `-Tdata` and `-Ttext`. For example, consider a dummy library which is linked by the following command:

```
$ ld -o libdummy.so -shared
        dummy.o  -Tdata 0x20000000
```

This causes the load address of the text segment of `libdummy.so` to be `0x00000000` and the load address of data segment to be `0x20000000`, creating a gap of size `0x20000000`. Assuming the text segment is mapped at address `0x40014000` (Note: addresses from `40000000` to `40014000` are used by the dynamic linker itself: `/lib/ld-2.2.5.so`), the data segment will be mapped at address `0x60014000`, thereby offsetting the base address of `/lib/i686/libc.so.6`.

The second approach does not provide the advantage of having a freshly randomized base address for each invocation of the program, but does have the benefit that it requires no changes to the loader or rest of the system. We have used this approach in our implementation. With this approach, changing the starting address to a different (random) location requires the library to be re-obfuscated (to change its preferred starting address).

2.3.3 Text/data segment randomization

Relocating a program's text and data segments is desirable in order to prevent attacks which modify a static variable or jump to existing program code. The easiest way to implement this randomization is to convert the program into a shared library containing position-independent code, which, when using `gcc`, requires compiling with the flag `-fPIC`. The final executable is created by introducing a new `main` function which loads the shared library generated from the original program (using `dlopen`) and invokes the original `main`. This allows random relocation of the original program's text and data segments. However, position-independent code is less efficient than its absolute address-dependent counterpart, introducing a modest amount of extra overhead.

An alternative approach is to relocate the program's code and data at link-time. In this case, the code need not be position-independent, so no performance overhead is incurred, Link-time relocation of the starting address of the executable can be accomplished by simple modifications to the scripts used by the linker.

Our implementation supports both of these approaches. Section 4 presents the performance overheads we have observed with each approach.

2.3.4 Random stack frame padding

Introducing padding within stack frames requires that extra storage be pushed onto the stack during the initialization phase of each subroutine. There are two basic implementation issues that arise.

The first issue is the randomization of the padding size, which could be static or dynamic. Static randomization introduces practically no runtime overhead. Dynamic randomization requires the generation of a random number at regular intervals. Additionally, the amount of extra code required for each function preamble is significant. Moreover, if the randomization changes the distance between the base of the stack frame and any local variable (from one invocation of a function to the next) then significant changes to the code for accessing local variables are required, imposing even more overheads. For these reasons, we have currently chosen to statically randomize the padding, with a different random value used for each routine.

The second issue concerns the placement of the padding. As shown in Figure 3, there are two basic choices: (1) between the base pointer and local variables, or (2) before parameters to the function:

1. *Between the base pointer and local variables.*
   This requires transformation of the callee to modify the instruction which creates the space for the local variables on the stack. Local variables are accessed using instructions containing fixed constants corresponding to their offset from the base pointer. Given that the padding is determined statically, the transformation simply needs to change the constants in these instructions. The main benefit of this approach is that it introduces a random gap between local variables of a function and other security-critical data on the stack, such as the frame pointer and return address, and hence makes typical stack-smashing attacks difficult.
2. *Before parameters to the function.*
   This is done by transforming the caller. First, the set of argument-copying instructions is located (usually `PUSH` instructions). Next, padding code is inserted just before these instructions. The primary advantage of this approach is that the amount of padding can change dynamically. Disadvantages of the approach are (a) in the presence of optimization, the argument-pushing instructions may not be contigu-

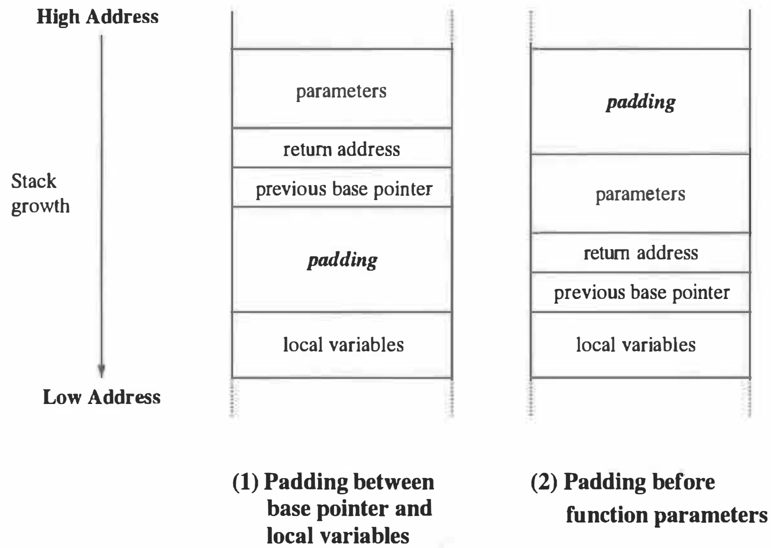


Figure 3: Potential locations of padding inserted between stack frames.

ous, which makes it difficult to determine where the padding is to be introduced, and (b) it does not make stack-smashing attacks any harder since the distance between the local variables and return address is left unchanged.

We have implemented the first option. As mentioned earlier, extraction of accurate control-flow graphs can be challenging for some routines. To ensure that our transformation does not lead to an erroneous program, the following precautions are taken:

- Transformation is applied to only those routines for which accurate control-flow graphs can be extracted. The amount of padding is randomly chosen, and varies from 0 to 256, depending on the amount of storage consumed by local variables, and the type of instructions used within the function to access local variables (byte- or word-offset). From our experience on instrumentation of different binaries, we have found that around 95 – 99% of the routines are completely analyzable.
- Only functions which have suitable behavior are instrumented. In particular, the function must have at least one local variable and manipulate the stack in a standard fashion in order to be instrumented. Moreover, the routines should be free of non-standard operations that reference memory using relative addressing with respect to the frame pointer.
- Only *in place* modification of the code is performed. By *in place*, we mean that the memory layout of the routines is not changed. This is done in order to avoid having to relocate the targets of any indirect calls or jumps.

These precautions have limited our approach to instrument only 65% to 80% of the routines. We expect that this figure can be improved to 90+% if we allow modifications that are not in-place, and by using more sophisticated analysis of the routines.

2.3.5 Heap randomization

The base address of the heap can be randomized using a technique similar to the stack base address randomization. Instead of changing the stack pointer, code is added to allocate a randomly-sized large chunk of memory, thereby making heap addresses unpredictable. In order to randomize the relative distances between heap data, a wrapper function is used to intercept calls to `malloc`, and randomly increase the sizes of dynamic memory allocation requests by 0 to 25%. On some OSes, including Linux, the heap follows the data segment of the executable. In this case, randomly relocating the executable causes the heap to also be randomly relocated.

3 Effectiveness

Address obfuscation is not a bulletproof defense against all memory error exploits, but is instead a probabilistic technique which increases the amount of work required before an attack (or sequence of attacks) succeeds. Hence, it is critical to have an estimate of the increase in attacker work load. In this section, we first analyze the effectiveness of address obfuscation against previously reported attacks and attack variations ("classic" attacks). Then we discuss attacks that can be specifically crafted to exploit weaknesses of address obfuscation.

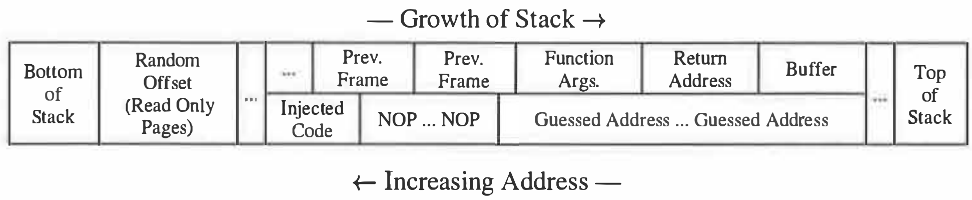


Figure 4: Format of an attack which uses a large buffer overflow to increase the odds of success.

3.1 Classic Attacks

Address obfuscation provides good protection against the majority of the "classic" attacks. Most of these attacks involve overwriting of a single pointer or datum without any ability to read the memory contents before attacking. Against address obfuscation, an attacker is forced to make guesses about the address of one or more program values in order to succeed.

3.1.1 Stack Smashing Attacks

A classic stack-smashing attack is absolute address-dependent, since the absolute address of the injected code must be placed in the return address stored in the stack frame. Let N be the size of the virtual address space available for the initial random stack offset, and assume that the stack offset is chosen randomly from $\{0 \ldots N-1\}$ (with a uniform distribution). Furthermore, we don't wish to allow an offset of zero, and Linux requires that the stack pointer be a 32-bit word-aligned address, which reduces the set of possible offsets to $\{4, 8, \ldots N\}$. (In this analysis, we assume that the one-time offset N is much larger than the effect of stack-frame padding, and hence ignore the latter. The purpose of stack-frame padding is to introduce significant additional randomization into the addresses so that attacks become difficult even if an attacker has somehow learned the value of N.)

Assuming the attacker knows the value of N, the attacker can guess an address randomly and have a $\frac{4}{N}$ chance of success. Moreover, if the guess happens to be wrong, then the program will likely crash, and will have to be restarted. At this time, a new random value for stack offset will be generated, which means that each failure does not provide any information to the attacker. Thus, the probability of a successful attack after k attempts is given by $1-(1-\frac{4}{N})^k$. From this, it can be shown that the probability of success approaches 0.5 after about $\frac{N}{8}$ attempts.

The attacker can improve the odds of success by increasing the size of the attack data. This can be done by writing to the buffer a block containing copies of a guessed address G (enough copies to be relatively sure that the return address is overwritten — in our implementation, of the order of 256 copies), followed by a block of K NOPs, and then the attack code. As long as G falls somewhere in the block of NOPs (or directly equals the first instruction of the inject code), the attack will succeed. This is illustrated in Figure 4, which shows the overlap between the stack values (along the top), and the attack data (along the bottom). When the current function returns, execution will jump to the guessed address G, which the attacker hopes will be within the range of the NOPs or the first instruction of the injected code.

The insertion of K NOPs increases the odds of success by a factor of K to $\frac{4 \cdot K}{N}$, reducing the average number of attempts for a reasonable chance of success to roughly $\frac{N}{8 \cdot K}$. Fortunately, K is limited in size because the attacker must avoid writing to the read-only stack padding. If the overflow runs into the read-only region, a segmentation fault will occur, preventing the attack from succeeding. This restricts the value of K to be much smaller than N. C programs tend not to use too much stack space; in the example programs of Figure 5, the amount of average stack storage allocated ranged from 1 to 4 kilobytes. For such programs, the maximum ratio of N to K will be $2.5 \cdot 10^4$, and the odds of a single attack succeeding will be $\frac{4}{2.5 \cdot 10^4}$, resulting in about 3000 attempts, or 12 megabytes of data transmitted, for a reasonable (≈ 0.5) probability of success. While this may seem like a small number, note that:

- every failure will cause a branch to a random address, which is highly likely to cause the target program to crash, so an attacker is not simply free to keep trying different addresses until an attack attempt succeeds. Instead, the repeated crashing of the program is likely to raise suspicion of the intruder's presence.
- the total amount of data that needs to be sent by the attacker is obtained by multiplying the size of attack data by the number of attack attempts. This number will be of the order of $\frac{N}{8}$, and is largely independent of the size of data used in each attack attempt.

3.1.2 Existing code attacks

Existing code attacks, also called *return-into-libc* attacks, typically involve overwriting the return address

on the stack with the address of existing code, typically a function in the standard C-library, such as `execve`. The arguments to this function will be taken from the stack, which has been overwritten by the same buffer overflow to contain the data chosen by attacker. In order for such an attack to succeed, the attacker needs to guess the location of the vulnerable function. With a randomization of the order of 100MB, and given the constraint that the base addresses of libraries and the executable must start at a multiple of page size (4KB), the probability of success is of the order of $4 \cdot 10^{-5}$.

Attacks that corrupt other stack-resident function pointers are all similar to an existing code attack, and the probability of a successful attack remains the same as with existing code attacks.

3.1.3 Format-String Attacks

A format-string vulnerability [33] occurs whenever a program contains a call to the `printf` family of functions with a first parameter (format string) that is provided by an attacker. Since the format string provides a great deal of control over the behavior of `printf` function, the ability of an attacker to provide a format string can be likened to the ability to execute attacker-chosen code. For this reason, most techniques developed to deal with buffer overflows are not effective against format string attacks.

The common form of this attack uses the somewhat obscure `%n` format parameter, which takes a pointer to an integer as an argument, and writes the number of bytes printed so far to the location given by the argument. The number of bytes printed can be easily controlled by printing an integer with a large amount of padding, e.g., `%432d`. The `printf` function assumes that the address to write into is provided as an argument, i.e., it is to be taken from the stack. If the attacker-provided format string is stored on the stack, and if printf can be tricked into extracting arguments from this portion of the stack, then it is possible for an attacker to overwrite an arbitrary, attacker-specified location in memory with attacker-specified data. Such an attack can be used to change return values without trampling over canary values used by StackGuard and other approaches.

The format-string attack described above is an absolute-address dependent attack. It requires the attacker to know the absolute location where the return address is stored on the stack, and the absolute location where the attack code is present. This means that the probability of a successful attack using this approach cannot be any larger than that for stack-smashing attacks.

Certain kinds of format-string vulnerabilities can be exploited to read stack contents. In particular, if the vulnerable `printf` (or variant) call is one that sends its output to the attacker, then the attacker can potentially learn the randomizations used in the program, and use this knowledge to craft a successful attack. (See Section 3.2.1 for details.)

3.1.4 Data Modification Attacks

Attacks which target non-pointer data values are one of the most difficult to defend against. For instance, a string which contains a shell command may be stored adjacently to the end of a buffer with an overflow vulnerability. In this case, an attacker can overflow the buffer with ASCII text containing a different command to be executed. The success of the attack depends only upon the relative distance between the buffer and the command string. Furthermore, even if the relative distance is randomized, the attacker can use blank characters as padding to increase the odds of success. If the attacker pads the injected string with more blanks than the maximum increase in distance between the buffer and the shell string, then the odds of success are high, especially when the data is located in the static area. If it is located on the stack, then the introduction of blanks (or other padding characters) may corrupt critical data on the stack, which may cause the program to crash. For this reason, such padding may not be very successful for stack-resident data.

Our current implementation provides limited protection against this attack, in the case where the data resides on the stack or heap. In the case of heap, if the overflow attack overwrites critical data within the same malloc-ed block as the target of the copy operation, then randomization does not help. Otherwise malloc randomization is effective, with the effectiveness increasing proportionately with the number of malloc blocks that are overwritten by the attack. Similarly, if the buffer and vulnerable data appear on the same stack frame, our current implementation does not provide any help. However, if they reside in different stack frames, then some level of protection is available, depending on the distance between the buffer and the vulnerable data.

The scope of protection can be expanded using the technique presented in [16], where all of the sensitive data (such as function and data pointers) can be located at addresses below the starting address of any buffer. Since the overflows can only move upward in memory, they can never reach from the buffer to a sensitive data location without crossing over into previous stack frames, in which case the return address will be corrupted.

Our current implementation provides no protection against relative address-dependent overflows that corrupt data in the static area. A fuller implementation of address obfuscation, which includes reordering of static variables as well as padding between them, will indeed

provide a good degree of protection against data modification attacks in the static area.

3.1.5 Heap Overflow and Double-Free Attacks

Due to the lack of adequate checking done by `malloc` on the validity of blocks being freed, code which frees the same block twice corrupts the list of free blocks maintained by `malloc`. This corruption can be exploited to overwrite an arbitrary word of memory with an arbitrary value [2]. A heap overflow attack achieves the same effect through a buffer overflow that also corrupts the data structures maintained by `malloc` [23].

Both of these are absolute address-dependent attacks, and the protection provided by address obfuscation is quite good, as the address of a single word is randomized over $\frac{10^8}{4}$ possible values.

3.1.6 Integer Overflow Attacks

Integer overflow attacks exploit an integer overflow to bypass runtime checks in a program. Since an integer has a fixed size, an overflow during a computation causes it to change its value in an undefined manner (typically, the value "wraps around" from a large positive value to a small negative one, or vice-versa). Due to the wrap-around, boolean conditions which test the values of integers resulting from arithmetic overflow are often incorrectly evaluated. For example, if i is sufficiently large, the expression $i + 5$ can overflow, resulting in a negative value, and causing the condition $i + 5 > limit$ to evaluate to false, when it should be true. This effectively disables the bounds checking, allowing an overflow attack to be performed in spite of the bounds checking.

The level of protection provided by address obfuscation from these kinds of attack is the same as for normal buffer overflow attacks. In particular, if the target corrupted by an attack is a pointer, then the probability of a successful attack is low. This was the case with the recent Snort integer overflow vulnerability. If the attack targets security critical data, then the protection is similar to that for relative address attacks. In particular, a good degree of protection is available for heap-resident data, while the level of protection for stack resident data is some what lesser. As an example, the sshd integer overflow attack involved overwriting a critical piece of string data with a null character, which was interpreted by the sshd server to mean that no password was required for a user to log in. Address obfuscation provides a good degree of protection against such an attack, while some of the related approaches such as PointGuard can be defeated by this attack.

3.2 Specifically Crafted Attacks

We have identified three specific attacks which can be used to attempt to defeat address obfuscation when the victim program contains the "right" vulnerability. These occur when (1) a program has a bug which allows an attacker to read the memory contents, or (2) an overflow exists that can be used to modify two pointer values (a buffer pointer and a function pointer), or (3) an overflow can be used to overwrite just the lower part of a pointer. In the case of (1), the attacker can craft an attack that succeeds deterministically. In the case of (2) and (3), the probability of success is significantly higher than the classic attacks, but far from deterministic.

We note all of the attacks discussed in this section require vulnerabilities that are very uncommon. Moreover, although our current implementation is vulnerable to these attacks, a full implementation of address obfuscation, employing all of the transformations described in Section 2.1, and using dynamically changing random values, will be much less vulnerable.

3.2.1 Read/Write Attacks

If a program contains a bug which allows an attacker to print the values stored in arbitrary memory locations, then most of the existing security schemes can be compromised if there is a vulnerability somewhere in the program. In the case of address obfuscation, the attacker can compare pointer values stored in the program against a local, non-obfuscated copy, and possibly decipher the obfuscation mapping. A specific instance of this occurs when an attacker can control the format-string passed to a `printf`, provided the vulnerable print statement sends its output to the attacker [29]. Given such a vulnerability, an attacker can send a format string that would cause the stack contents to be printed. From the output, the attacker can guess with a high probability (or with certainty, if no stack frame padding is used) the locations holding saved frame pointer and return address. By comparing these values with those that can be observed on their local version of the vulnerable program that has not been obfuscated, the attacker can identify the obfuscation mapping. Armed with this mapping, the attacker can develop an attack that will succeed with a high probability. This time, the attacker will use the standard format-string attack that uses the `n%` directive.

We point out that changing just the base addresses of different memory regions, as done with PaX ASLR, does not help with this attack. Most other techniques, such as PointGuard and StackGuard are also vulnerable to this attack. In the case of PointGuard, the obfuscated stack can be compared to a non obfuscated process, and the xor mask value can be inferred. In the case of StackGuard, the stack can be examined to determine the ca-

nary value, and then stack smashing can be used.

Address obfuscation, as implemented now, seems to provide some additional protection over ASLR: it is no longer possible to deterministically identify the location of frame pointer or return address. But this added difficulty does not translate into additional protection: the format-string based read attack does not cause the program to crash, so the attacker can perform multiple attacks to read the stack multiple times until he/she can determine the frame pointer with certainty. However, if the stack-frame padding is varied continuously at runtime, then address obfuscation will provide significant degree of protection. In this case, the location of the buffer, the saved frame pointer, as well as the return address, will change between the time the attacker read the contents of the stack and the time he/she tries to modify the return address. This will significantly decrease the chances of a successful attack. Probability of a successful existing-code attack can also be decreased significantly by using the more general form of address obfuscation of code, which involves reordering routines, etc.

3.2.2 Double Pointer Attacks

A program which contains a both a (preferably stack-allocated) pointer to a buffer and a buffer overflow vulnerability can be exploited to work around obfuscation. For example, consider the following code fragment, which is similar to one suggested for defeating StackGuard [7]:

```
void
f(char *user_input1, char *user_input2) {
  char *buf1 = malloc(100);
  char buf2[100];
  strcpy(buf2, user_input1);
  strncpy(buf1, user_input2, 100);
  ...
```

The steps required to exploit this code are as follows. First, the attacker can guess an address G likely to be valid (somewhere in the heap is a good choice). Second, the first `strcpy` to `buf2` can be overflowed to overwrite the the top stack locations with G, setting both `buf1` and the saved return address to equal G. Once this is done, the `strcpy` to `buf1` will copy `user_input2` to G. `user_input2` should contain the injected code. When the function returns, it will jump to address G, which is the start of the code injected via `user_input2`.

The probability of success with this attack is proportional to the probability of guessing a valid address G in memory. This probability is small for programs that use small amounts of memory as compared to the amount of randomization. For instance, if the program uses a megabyte of memory, then the probability success (with a 100MB padding) is one in a hundred. The same line of reasoning holds with PointGuard: the attacker can overwrite `buf1` and the return address with G, but these values will be interpreted as G xor M where M is the xor mask used by PointGuard to encrypt pointers. This means that the probability of success is proportional to that of guessing a G such that G xor M corresponds to a writable portion of the memory. This probability is given by (size of data memory used by program)/(size of address space), a quantity that is smaller than the corresponding number for address obfuscation.

3.2.3 Partial Overwrite Attacks

A *partial overwrite attack* is an attack which overwrites only part of a targeted value. For example, under the x86 architecture, an overflow could overwrite just the least significant byte of the return address. (This is hard to achieve if the buffer overflow was the result of an unchecked `strcpy` or similar function, since the terminating null character would clobber the rest of the return address. Thus, we need a buffer overflow that does not involve strings.) Since the only transformation made to code addresses is that of changing the base address, and since the quantity of change is constrained to be a multiple of the page size (4096 bytes on Linux), the location pointed by the return address is predictable when we we change its last 8 bits.

If exploitable code (i.e., code that can be used as a target in the case of existing code attacks) can be found within 256 bytes of the return address of a function with buffer-overflow vulnerability, then this attack will work against address obfuscation. However, it is very unlikely that such exploitable code can be found, so the attack suggested in [3] is more elaborate. Specifically, the attack involves the use of a call to the `printf` function in the caller code that precedes the call to the function with buffer overflow vulnerability. The attack then modifies the return address so that a return goes to the instruction that calls `printf`. The argument of the vulnerable function, which was attacker-provided, now becomes the argument to `printf`. At this point, the attacker can print the contents of the stack and then proceed as with the case where a format string bug allowed the attacker to read the stack contents.

Note that the stack-frame padding significantly increases the difficulty of carrying out this attack. In particular, there is a significant level of uncertainty (of the order of 128 bytes) in the distance between the vulnerable buffer and the return address, which the attacker can overcome only through guessing. If additional code address obfuscation transformations are used, (for instance, reordering of routines or introducing gaps within routines) then the attack becomes even harder.

Program	Combination (1)		Combination (2)	
	% Overhead	Standard Deviation (% of mean)	% Overhead	Standard Deviation (% of mean)
`tar`	-1	3.4	0	5.2
`wu-ftpd`	0	1.4	2	2.1
`gv`	0	6.1	2	7.1
`bison`	1	2.0	8	2.3
`groff`	-1	1.1	13	0.7
`gzip`	-1	1.9	14	2.5
`gnuplot`	0	0.9	21	1.0

Figure 5: Performance overhead introduced by address obfuscation.

4 Performance

We have collected performance data on the implementation of randomization of different memory regions. The following randomizations were implemented:

- relocating the base of the stack, heap, and code regions
- introduction of random gaps within stack frames, and at the end of memory blocks requested by malloc. The stack frame gaps were determined statically for each routine, while the malloc gaps can change with each malloc request.

We studied two different approaches for randomizing the start address of the executable:

- *Combination 1:* static relocation performed at link-time.
- *Combination 2:* dynamic relocation performed at load-time.

Both approaches incorporate all of the transformations mentioned above. Note that dynamic relocation requires the executable be compiled into position-independent code, which introduces additional runtime overheads.

Figure 5 shows the performance overheads due to the two combinations of transformations. All measurements were taken on an 800 MHz, Pentium III, 384 MB RAM machine with Red Hat 7.3 Linux OS. Average execution (system + user) time was computed over 10 runs. The overheads measured were rounded off to the nearest integral percentage. (Further precision was meaningless, given the standard deviations shown in the table.)

From the table, we see that combination (1) incurs essentially no runtime overhead (note that the negative overheads are below the standard deviation and are hence not statistically significant).

Combination (2) has noticeable runtime overhead. This is because it requires position-independent code, which is less efficient, since it performs extra operations before every procedure call, and every access to static data. On the other hand, when code is already being distributed in DLL form, combination (2) provides broad protection against memory error exploits without any additional overhead.

5 Related Work

5.1 Runtime Guarding Against Stack-Smashing and Format String Attacks

These techniques transform or augment a program to protect the return address or other specific values from being overwritten. *Stackguard* [11] is a modified version of the *gcc* compiler in which the generated code places *canary values* around the return address at runtime, so that any overflow which overwrites the return address will also modify the canary value, enabling the overflow to be detected. *StackShield* [6] and *RAD* [9] are based upon a similar modification to the compiler, but keep a separate copy of the return address instead of using canary values. *Libsafe* and *Libverify* [6] are dynamically loaded libraries which provide protection for the return address without requiring recompilation. Etoh and Yoda [16] use a source-code transformation approach which uses both canary values and relocates stack-allocated arrays so that they cannot overflow into local variables. *FormatGuard* [12] transforms source code using a modified version of *cpp* (the C Preprocessor) combined with a wrapper function for the `printf` function, so that format-string attacks are detected at runtime.

While these techniques are useful for guarding against specific attacks, their drawback is that they can deal with only a small subset of the total set of memory exploits shown in Figure 2.

5.2 Runtime Bounds and Pointer Checking

These techniques prevent buffer overflows by checking each memory access operation that can potentially cause a memory error to ensure that it does not happen. Approaches used to insert the required checks have included source-to-source translation [25, 5], specially

modified compilers [36, 22], binary rewriting [19], and virtual machines/interpreters [24]. All of the above techniques currently suffer from significant drawbacks: runtime overheads that can often be over 100%, restriction to a subset of C-language, and changes to the memory model or pointer semantics. In contrast, the focus of this paper is on techniques that produce very low overheads and are fully compatible with all C-programs.

5.3 Compile-Time Analysis Techniques

Compile-time analysis techniques [18, 32, 37, 14, 26] analyze a program's source code to determine which array and pointer accesses are safe. While these approaches are a welcome component of any programmer's debugging arsenal, they generally suffer from one or more of the following shortcomings: they do not detect all memory errors, they generate many false positive warnings, and/or they do not scale to large programs. The focus of our work is the development of techniques that require no additional effort on the part of programmers, and hence can be applied to the vast base of existing software, in binary form, with no programmer effort.

Hybrid approaches perform runtime memory-error checking, but also use static analysis to minimize the number of checks. *CCured* [28] and Cyclone [21] are two recent examples of this approach. One difficulty with these approaches is that they are not 100% compatible with existing C-code. Moreover, they disable explicit freeing of memory, and rely on garbage collection.

5.4 Code Obfuscation

Code obfuscation [38, 10, 4] is a program transformation technique which attempts to convolute the low-level semantics of programs without affecting the user-observable behavior, making obfuscated programs difficult to understand, and thereby difficult to reverse-engineer. The key difference between program obfuscation and address obfuscation is that program obfuscation is oriented towards preventing most static analyses of a program, while address obfuscation has a more limited goal of making it impossible to predict the relative or absolute addresses of program code and data. Other analyses, including reverse compilation, extraction of flow graphs, etc., are generally not affected by address obfuscation.

5.5 Randomizing Code Transformations

As mentioned earlier, address obfuscation is an instance of the broader idea of introducing diversity in nonfunctional aspects of software, an idea suggested by Forrest, Somayaji, and Ackley [17]. Their implementation model was called a *randomizing compiler,* which can introduce randomness in several non-functional aspects of the compiled code without affecting the language semantics. As a proof of concept, they developed a modification to the gcc compiler to add a random amount of padding to each stack allocation request. This transformation defeats most stack-smashing attacks prevalent today, but does not work against the large overflow attacks of the sort described in Section 3.

In the past year or two, several researchers [8, 1, 39, 15] seem to have independently attempted to develop randomization as a practical approach to defeat buffer-overflow and related attacks. Work by Chew and Song [8] randomizes the base address of the stack, system call numbers, and library entry points, through a combination of program loader modifications, kernel system call table modifications, and binary rewriting. Xu, Kalbarczyk, and Iyer developed *transparent runtime randomization* [39], in which the Linux kernel is modified to randomize the base address of stack, heap, dynamically loaded libraries, and GOT. The PaX project's *address space layout randomization* (ASLR) approach [1] randomizes the base address of each program region: heap, code, stack, data. Of these, the ASLR approach is the most advanced in terms of its implementation. As noted earlier, ASLR is vulnerable to attacks that rely on adjacency information such as the relative addresses between variables or code, and attacks that can provide information about the base addresses of different memory segments. The introduction of additional randomization in address obfuscation, in the form of random-sized gaps within stack frames and blocks allocated by `malloc`, reordering of (and random padding within) code and static variables, can address these weaknesses. Another important difference between the above works and ours is that our obfuscations are implemented using program transformations, whereas the other works are implemented using operating system modifications. For this reason, our approach can be more easily ported to different operating systems. Moreover, it can protect individual (security-critical) applications without having to make any changes to the rest of the system.

The PointGuard [13] approach complements ours in that it randomizes ("encrypts") stored pointer values, as opposed to the locations where objects are stored. The encryption is achieved by xor'ing pointer values with a random integer mask generated at the beginning of program execution. It shares many of the benefits (such as broad protection against a wide range of pointer-related attacks) and weaknesses (susceptibility to attacks that read victim process memory to identify the mask). The principal differences are that (a) PointGuard does not protect against attacks that do not involve pointer values,

e.g., attacks that modify security-critical data through a buffer overflow, and (b) probability of successful attacks is smaller with PointGuard than with address obfuscation since the range of randomization can be as large as the address space. It should also be noted that PointGuard is dependent on the availability of accurate type information. Many C-language features, such as the ability to operate on untyped buffers (e.g., `bzero` or `memcpy`), functions that take untyped parameters (e.g., `printf`), unions that store pointers and integer values in the same location, can make it difficult or impossible to get accurate type information, which means that the corresponding pointer value(s) cannot be protected.

6 Conclusion

We believe that address obfuscation has significant potential to constrain the increasing threat of widely spread buffer overflow-type of attacks. By randomly re-arranging the memory space that holds a computer program and its data during execution, the core vulnerability that buffer overflow attacks have been exploiting is addressed — namely, the predictable location of control information and critical data. Unlike many existing techniques, which deploy attack-specific mechanisms to overcome known attack scenarios, address obfuscation is a generic mechanism that has a broad range of application to many memory error-related attacks.

Since each system is obfuscated differently, even if an attacker successfully subverts one system on a network, the attack will have to essentially start over from scratch and make many attempts before a second system can be subverted. In the context of self-replicating attacks, this factor will greatly slow down the spread of worms and viruses. Thus, address obfuscation provides a simple and effective solution to combat the spread of viruses and worms which replicate by exploiting memory errors.

Our main goal for the future is to improve the quality of randomization that can be done at the binary level. In particular, we are interested in randomizing the relative distances between objects in all regions of a program, instead of just the stack and heap, as is the case with our current implementation. There are basically two avenues for this work. The first is a tool that works with existing binary files. Such a tool will be restricted in the types of obfuscations which can be applied, but will have a wide potential impact. Addressing relative-distance issues requires both inserting padding between and permuting the order of data and code, which requires the relocation of affected addresses. Performing these sorts of relocations on binaries is not always feasible due to the difficulty of distinguishing pointers from non-pointers, sizes of data objects, and code from data. We plan on developing better analysis tools, and combining this with a flexible transformation strategy that applies as many obfuscations as possible within the limits of the analysis.

The second avenue is to augment binaries with an extra section that contains the information required to safely perform relocations. This approach requires a relatively minor change to the compiler infrastructure, as the information required is similar to the information already being generated to support the linking of object modules. Given the extra information, a program can be obfuscated at link- or load-time in a more thorough manner which will change all relative and absolute addresses in every program region.

Acknowledgments

This research was supported in part by AFOSR grant F49620-01-1-0332, ONR University Research Initiative Grant N00140110967, and NSF grants CCR-0098154 and CCR-0208877.

References

[1] Pax. Published on World-Wide Web at URL http://pageexec.virtualave.net, 2001.

[2] Anonymous. Once upon a free .... *Phrack*, 11(57), August 2001.

[3] Anonymous. Bypassing pax aslr protection. *Phrack*, 11(59), July 2002.

[4] D. Aucsmith. Tamper-resistant software: An implementation. In Ross Anderson, editor, *Information hiding: first international workshop, Cambridge, U.K., May 30–June 1, 1996: proceedings*, volume 1174 of *Lecture Notes in Computer Science*, pages 317–333, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1996. Springer-Verlag.

[5] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI)*, pages 290–301, Orlando, Florida, 20--24 June 1994. *SIGPLAN Notices* 29(6), June 1994.

[6] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the 2000 USENIX Annual Technical Conference (USENIX-00)*, pages 251–262, Berkeley, CA, june 2000.

[7] Bulba and Kil3r. Bypassing stackguard and stackshield. *Phrack*, 11(56), May 2000.

[8] Monica Chew and Dawn Song. Mitigating buffer overflows by operating system randomization. Technical Report CMU-CS-02-197, Carnegie Mellon University, December 2002.

[9] Tzi-cker Chiueh and Fu-Hau Hsu. Rad: A compile-time solution to buffer overflow attacks. In *21st International Conference on Distributed Computing*, page 409, Phoenix, Arizona, April 2001.

[10] Christian Collberg, Clark Thomborson, and Douglas Low. Breaking abstractions and unstructuring data structures. In *Proceedings of the 1998 International Conference on Computer Languages*, pages 28–38. IEEE Computer Society Press, 1998.

[11] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, jan 1998.

[12] Crispin Cowan, Matt Barringer, Steve Beattie, and Greg Kroah-Hartman. Formatguard: Automatic protection from printf format string vulnerabilities. In *USENIX Security Symposium*, 2001.

[13] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, Washington, D.C., August 2003.

[14] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Cleanness checking of string manipulations in C programs via integer analysis. In *Static Analysis Symposium*, volume 2126 of *Lecture Notes in Computer Science*, pages 194–?? Springer Verlag, June 2001.

[15] Daniel C. DuVarney, R. Sekar, and Yow-Jian Lin. Benign software mutations: A novel approach to protect against large-scale network attacks. Center for Cybersecurity White Paper (prepared for Airforce Office of Scientific Research), October 2002.

[16] Hiroaki Etoh and Kunikazu Yoda. Protecting from stack-smashing attacks. Published on World-Wide Web at URL http://www.trl.ibm.com/projects/security/ssp/main.html, June 2000.

[17] Stephanie Forrest, Anil Somayaji, and David H. Ackley. Building diverse computer systems. In *6th Workshop on Hot Topics in Operating Systems*, pages 67–72, Los Alamitos, CA, 1997. IEEE Computer Society Press.

[18] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *ACM SIGPLAN Conference on Programming Language and Design*, Atlanta, GA, May 1999.

[19] Reed Hastings and Bob Joyce. Purify: A tool for detecting memory leaks and access errors in C and C++ programs. In USENIX Association, editor, *Proceedings of the Winter 1992 USENIX Conference*, pages 125–138, Berkeley, CA, USA, January 1992. USENIX.

[20] Oded Horovitz. Big loop integer protection. *Phrack*, 11(60), December 2002.

[21] Trevor Jim, Greg Morrisett, Dan Grossman, Micheal Hicks, James Cheney, and Yanling Wang. Cyclone: a safe dialect of C. In *USENIX Annual Technical Conference*, Monterey, CA, June 2002.

[22] Robert W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In M. Kamkar and D. Byers, editors, *Third International Workshop on Automated Debugging*. Linkoping University Electronic Press, 1997.

[23] Michel Kaempf. Vudo malloc tricks. *Phrack*, 11(57), August 2001.

[24] Stephen Kaufer, Russell Lopez, and Sesha Pratap. Saber-C — an interpreter-based programming environment for the C language. In USENIX Association, editor, *Summer USENIX Conference Proceedings*, pages 161–171, Berkeley, CA, USA, Summer 1988. USENIX.

[25] Samuel C. Kendall. Bcc: run–time checking for c programs. In *Proceedings of the USENIX Summer Conference*, El. Cerrito, California, USA, 1983. USENIX Association.

[26] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, Washington, D.C., August 2001.

[27] Mudge. How to write buffer overflows. Published on World-Wide Web at URL http://www.insecure.org/stf/mudge_buffer_overflow_tutorial.html, 1997.

[28] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *Symposium on Principles of Programming Languages (POPL '02)*, pages 128–139, Portland, OR, January 2002.

[29] Nergal. The advanced return-into-lib(c) exploits. *Phrack*, 11(58), Dec 2001.

[30] Mary Lou Nohr. *Understanding ELF Object Files and Debugging Tools*. Number ISBN: 0-13-091109-7. Prentice Hall Computer Books, 1993.

[31] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996.

[32] Radu Rugina and Martin Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation*, pages 182–195. ACM Press, 2000.

[33] scut. Exploting format string vulnerabilities. Published on World-Wide Web at URL http://www.team-teso.net/articles/formatstring, March 2001.

[34] Snort(tm) advisory: Integer overflow in stream4. April 2003. Published on World-Wide Web at URL http://www.kb.cert.org/vuls/id/JPLA-5LPR9S.

[35] Ssh crc32 attack detection code contains remote integer overflow. 2001. Published on World-Wide Web at URL http://www.kb.cert.org/vuls/id/945216.

[36] Joseph L. Steffen. Adding run-time checking to the portable c compiler. *Software-Practice and Experience*, 22:305–316, April 1992.

[37] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, San Diego, CA, 2000.

[38] Chenxi Wang, Jack Davidson, Jonathan Hill, and John Knight. Protection of software-based survivability mechanisms. In *International Conference on Dependable Systems and Networks*, Goteborg, Sweeden, July 2001.

[39] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent runtime randomization for security. Technical Report UILU-ENG-03-2207, Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, May 2003.

[40] Lu Xun. A linux executable editing library. Masters Thesis, 1999. available at http://www.geocities.com/fasterlu/leel.htm.

High Coverage Detection of Input-Related Security Faults

Eric Larson and Todd Austin

Advanced Computer Architecture Laboratory
University of Michigan
Ann Arbor, MI 48105
larsone@eecs.umich.edu, austin@eecs.umich.edu

Abstract

Improperly bounded program inputs present a major class of program defects. In secure applications, these bugs can be exploited by malicious users, allowing them to overwrite buffers and execute harmful code. In this paper, we present a high coverage dynamic technique for detecting software faults caused by improperly bounded program inputs. Our approach is novel in that it retains the advantages of dynamic bug detection, scope and precision; while at the same time, relaxing the requirement that the user specify the input that exposes the bug. To implement our approach, inputs are shadowed by additional state that characterize the allowed bounds of input-derived variables. Program operations and decision points may alter the shadowed state associated with input variables. Potentially hazardous program sites, such as an array references and string functions, are checked against the entire range of values that the user might specify. The approach found several bugs including two high-risk security bugs in a recent version of OpenSSH.

1. Introduction

Bugs in software can have a devastating effect in today's world. Computer viruses can exploit software bugs in order to run malicious code or gain access to restricted data. A highly visible and damaging example of this type of defect includes improperly bounded checks on network data. A common example of this class of defect is the buffer overflow. Input data is obtained from the network without checking to see if it will fit within a program buffer. Malicious users can exploit this bug by overwriting stack buffers in a way that overwrites the function return address to direct control to arbitrary code.

To prevent a buffer overflow exploit, it is necessary for the program to check input data to ensure it does not exceed the bounds of any buffer it may be used to reference. However, many programs either fail to check input data or check the data incorrectly. Such cases are often hard to find. For example, the code sequence in Figure 1 contains an off-by-one error. Such an error may be difficult to find if the programmer writing the code to check the reference is not aware that the index is incremented before it is referenced.

```
unsigned int x;
int array[5];
scanf("%d", &x);
if (x > 4) fatal("Index out of bounds");
x++;
a = array[x];
```

Figure 1: Example of an array bounds error. This code segment will overflow the array if x is 4.

Another common source of security bugs is improper use of the string library functions in C. Since the functions provide no checking, the responsibility resides with the programmer. To complicate matters, there is little consistency on how the different string functions operate. For instance, the `strcpy` command always copies a null character but `strncpy` will not copy the null character unless one is present within the specified limit. An example of a bug involving strings is shown in Figure 2. In this case, there is a check to filter out strings that are greater than 16 characters. However, the `strlen` command does not count the null character. If the source string is exactly 16 characters (not including the null character), it will pass the check though it contains 17 characters, including the null character. As a result, the null character does not get copied by the `strncpy` command, creating a potentially dangerous `strcpy` because the source is not null terminated. This type of problem is difficult to catch during testing since it requires a source string of exactly 16 characters. Also, the bug may not manifest in an error in the output when such an input is presented; this is likely the case if the character after the `temp` array happens to be a null.

```
char *bad_string_copy(char *src)
{
  char *dest;
  char temp[16];

  if (strlen(src) > 16) return NULL;
  strncpy(temp, src, 16);
  dest = (char *) malloc(16);
  strcpy(dest, temp);
  return dest;
}
```

Figure 2: Fault due to improper use of string library functions. If `src` has 16 characters (not including the null character), it will get copied into `dest` without a null character causing a problem in the subsequent `strcpy` function.

Code Segment	Value of x	Interval constraint on x
`unsigned int x;`		
`int array[5];`		
`scanf("%d", &x);`	2	$0 \le x \le \infty$
`if (x > 4) fatal("Index out of bounds");`	2	$0 \le x \le \infty$
`x++;`	2	$0 \le x \le 4$
`a = array[x];`	3	$1 \le x \le 5$ → ERROR!

Figure 3: Detecting an array bounds error. The error is detected even though the input value of 2 does not directly cause an error.

One approach to prevent security exploits is to add run-time support that will prevent malicious behavior. For example, a technique created by Lhee and Chapin [20] append type information to arrays and intercepts string functions to ensure the bounds of the array are not exceeded. The added run-time support remains with the program after it has been deployed resulting in a performance penalty. To avoid this penalty, software teams will sometimes employ testing efforts to detect errors at design-time, thereby reducing the need for run-time checking once the software has been deployed. Often, dynamic bug detection tools are used to aid validators by finding bugs that do not necessarily manifest in an error in the program output. These tools use run-time knowledge of all variables including pointers and variables that reside on the heap. Dynamic techniques can find defects over a wide scope of the program including bugs that span multiple function boundaries, library functions, or even process boundaries. However, they are limited to exposing only those defects that testers can expose. For example, the bug in Figure 1 would only be detected when the input value of four is provided. Without extensive testing, dynamic techniques are best used to expose defects in common-use scenarios.

In this work, we introduce a dynamic high coverage approach to detecting security faults caused by improperly bounded inputs. Our technique possesses the scope and program knowledge of a run-time technique while relaxing the requirement that the validator specify a set of inputs that exposes the defect. We implement our approach by shadowing all input values (and variables derived from input) with a state variable. State variables are introduced into the program when external inputs are read. External inputs encompass a variety of sources: command line arguments, input files, environment variables, and network data.

Integers are shadowed by an *interval constraint variable* that stores the lower and upper bounds of the range of values that the given variable may hold. They have an initial value indicating the input is unbounded with maximum range that can be represented by the data type of the variable. During execution, control tests and operators may narrow input interval constraints. Finally, at potentially dangerous uses of inputs, such as array references and trusted system calls, the entire range of an input value is validated using the computed interval constraint. As a result, all input-related faults are exposed for a given control path, even if the user-specified input did not directly expose the fault.

In Figure 3, we show how our technique can find the off-by-one error in the code segment from the example in Figure 1. In a conventional dynamic bug detection implementation, an error will not be detected unless `x` is four. In our correctness model, we improve defect coverage by extending input values with interval constraints. When the value is first read from input, it is given a range to span all possible values. At the control points (after the `if` statement in the example), the interval constraint can be narrowed because the value of `x` is now known to be ≤ 4. When the value of `x` is incremented, the interval is adjusted up by one on both ends. When the array access occurs, the interval of `x` is compared to the size of the array. Even though `x` has the legal value of two, an error is flagged since it is possible for the input to be five, which exceeds the bounds of the array.

Input strings are shadowed by state variables that hold the maximum possible size of the string and a flag that indicates if the string is known to contain a null character. Like integers, strings from external input sources are considered to have an unbounded maximum size. Control predicates that test the length of an input string can decrease the maximum string length. The null flag is initially set on input strings and is initially clear on uninitialized arrays. In order to set the null flag, a string must be copied in a manner that guarantees that null is set. String functions are checked to ensure that all strings passed as a parameter are null terminated and there is sufficient room in the destination string for copy operations. Our approach will verify these functions for all possible string lengths up to the maximum size making it unnecessary for a test to have the exact string length that can trigger an error.

The string example is revisited in Figure 4. Assume that the input to the function is a null terminated string consisting of 8 characters taken directly from input. Upon entry to the function, its maximum size will be unbounded. Though the user entered an 8 character string, it could have an entered a string of any length. Since the string has 8 characters it passes the check but its maximum size is reduced to 17. (We count the null character in our definition of string size.) In the `strncpy` function, the maximum size of 17 can exceed the available destination size limit of 16. Consequently, there is no guarantee that the null character is copied, and the null flag remains off for the array `temp`. This leads to an error when `temp` is used in the `strcpy` function

Code Segment	State for src	State for temp	State for dest
`char *bad_string_copy(char *src)`	max_sz: ∞, known_null: T		
`{`			
`char *dest;`			
`char temp[16];`		max_sz: 16, known_null: F	
`if (strlen(src) > 16) return NULL;`	max_sz: 17, known_null: T		
`strncpy(temp, src, 16);`		max_sz: 16, known_null: F	
`dest = (char *) malloc(16);`			max_sz: 16, known_null: F
`strcpy(dest, temp);`			ERROR (temp may not be null terminated)
`return dest;`			
`}`			

Figure 4: Detecting a string copy error. The error is detected because it is possible for the `strcpy` function to execute with a source that is not null terminated.

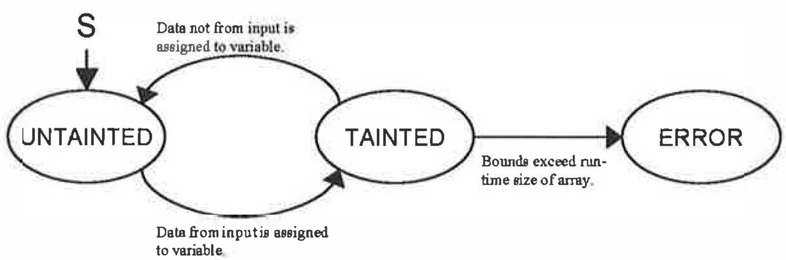


Figure 5: Program states for integer variables.

since it may not be null terminated. Even though the input string of 8 characters does not expose the error, our approach still detects it because all possible string lengths are verified. It is also worth noting that our approach would detect an error if the string `src` were directly copied into `dest` in the `strcpy` instead of using the `temp` array. In this case, the error would get signalled because the maximum size of the source (17 characters) is greater than the destination (16 characters).

Our approach is generic and can be applied to all programs. We have applied it to several programs and found a number of bugs including two major security bugs in a recent release of OpenSSH. It is portable and does not require any modifications to the source code. Unlike techniques that are designed to prevent malicious behavior while running, our technique is intended to be used to find faults before software is released. Given the degree of our analysis, the run-time performance impact of our technique is fairly high, making our approach only appropriate for the testing phase of development. Another limitation of our approach is that defect detection is control path sensitive. Good testing, however, can mitigate this effect by covering all of the interesting paths of a program.

The remainder of the paper is organized as follows. The next section describes our method for detecting input-related faults. Section 3 describes our methodology and our dynamic bug detection tool. Section 4 shows the types of bugs we have found using our approach and compares the run-time performance to an uninstrumented version of the program. Section 5 outlines related work, and Section 6 gives conclusions.

2. High coverage detection of software faults

Our high coverage scheme for detecting software faults centers on verifying all possible input values at dangerous points. At array references, the entire range of possible index values are checked to ensure the array bounds are not exceeded. This procedure is described in Section 2.1. Unsafe string functions such as `strncpy` are checked by assuming that input strings can have arbitrary length initially. String operations are checked to ensure there is sufficient room to store the largest possible string. Section 2.2 details our technique. Section 2.3 lists other situations we detect that lead to software faults.

2.1 Detecting dangerous array references

In order for an array reference to be considered safe, the index must be checked to determine if it is possible to exceed the bounds of the array. This is accomplished by attaching interval constraint information be to every variable that contains program input. To keep the maintenance of bounds information manageable, we identify at run-time which variables contain program input and only attach interval constraints to these variables. We start with a model of *tainted* data that is similar to that used in [1] and is summarized in Figure 5. Data that comes from unbounded input is considered tainted. Examples of unbounded input include environment variables, command line inputs, data read from files, and network packets. Tainted data is shadowed

Table 1: Sample integer rules and their effect on the bounds. `x'`, `y'`, and `a'` are tainted and `y` is untainted.

Rule	Operation	Input Interval Constraint
1	`a' = x'`	`a'.lb = max(MIN_VAL(a'), x'.lb)` `a'.ub = min(MAX_VAL(a'), x'.ub)`
2	`a' = x' + y`	`a'.lb = max(MIN_VAL(a'), x'.lb + y)` `a'.ub = min(MAX_VAL(a'), x'.ub + y)`
3	`a' = x' + y'`	`a'.lb = max(MIN_VAL(a'), x'.lb + y'.lb)` `a'.ub = min(MAX_VAL(a'), x'.ub + y'.ub)`
4	`a' = x' % y'`	`a'.lb = 0, a'.ub = max(abs(y'.lb), abs(y'.ub))`
5	`if (x' < y)`	`if (x' < y): x'.lb = x'.lb, x'.ub = min(x'.ub, y - 1)` `else: x'.lb = max(x'.lb, y), x'.ub = x'.ub`
6	`if (x' < y')`	`if (x' < y'): x'.lb = x'.lb, x'.ub = min(x'.ub, y'.ub - 1)` `y'.lb = max(y'.lb, x'.lb + 1), y'.ub = y'.ub` `else: x'.lb = max(x'.lb, y'.lb), x'.ub = x'.ub` `y'.lb = y'.lb, y'.ub = min(y'.ub, x'.ub)`
7	`if (x' == y)`	`if (x' == y): x'.lb = y, x'.ub = y` `else if (x'.lb == y): x'.lb = y + 1, x'.ub = x'.ub` `else if (x'.ub == y): x'.lb = x'.lb, x'.ub = y - 1` `else: x'.lb = x'.lb, x'.ub = x'.ub`
8	`if (x' == y')`	`if (x' == y'): x'.lb = y'.lb = max(x'.lb, y'.lb)` `x'.ub = y'.ub = min(x'.ub, x'.lb)` `else: x'.lb = x'.lb, x'.ub = x'.ub` `y'.lb = y'.lb, y'.ub = y'.ub`
9	`while (x' < y)`	`in loop: x'.lb = x'.lb, x.ub = min(x'.ub, y - 1)` `after loop: x'.lb = max(x'.lb, y), x.ub = x'.ub`

with interval constraint variables that track the lower and upper bounds for the variable. When an access to an array occurs, the bounds of the array index are compared with the run-time size of the referenced array. An error is declared if there is an index that can exceed the bounds of the array. When a variable is assigned a value that is not dependent on input (untainted), the destination variable is reset to the untainted state. Since only tainted variables need interval constraints, any variable transition from tainted to untainted will release shadow state.

Since arrays may only be indexed by an integer, only variables with integer type (`char`, `int`, `unsigned int`, etc.) can become tainted. When an integer variable becomes tainted, it is assigned upper and lower bounds based on the precision of the type. For unsigned integers, the lower bound is zero. Otherwise, it is the most negative value the variable can hold, based on type. Similarly, the upper bound is the largest possible value.

As tainted variables are operated upon or tested with control predicates, their interval constraints must be adjusted accordingly. Table 1 shows a list of representative operations and their effect on the upper and lower bounds of an interval constraint. In the table, ticked variables `a'`, `x'`, and `y'` refer to tainted variables while `y` represents an untainted variable. The notation `x'.lb` represents the lower bounds of tainted variable `x'`. The expressions `MIN_VAL(a)` and `MAX_VAL(a)` refer to the minimum and maximum values that `a` can have based on its type precision.

For simple assignment operations (Rule 1), the bounds are copied into the assigned value in most cases. However, it may be necessary to restrict the bounds if the size of the destination type is smaller than the size of the source type. This may also occur when assigning a signed value into an unsigned value. State propagations are also required when integers are passed at function calls, returned from functions, assigned within structures, or when copied by system functions such as `memcpy`.

Addition and other arithmetic operations adjust the bounds of the destination variable. In the first addition pattern (Rule 2), a tainted value is added to an untainted value. The bounds of the destination variable are computed by adding the run-time value of the untainted variable to the bounds of the tainted variable. If both variables are tainted (Rule 3), the bounds are added together to form the new worst-case bounds. Rule 4 singles out the modulus operator because the new range is strictly dependent on the value of the second operand. We also detect overflow situations; this is mentioned in more detail in Section 2.3.

Rules 5-9 narrow the input interval constraint based on knowledge gained from control predicates. In Rule 5, if the `if` condition is true, the upper bound is reduced to `y-1` unless the existing upper bound is already lower than `y-1`. If the condition is false, the lower bound must be at least `y`. Rule 6 refers to a situation where two tainted variables are compared to one another. If `x' < y'` is true, then no change is necessary to the lower bound of `x'` and the upper bound of `y'`. The upper bound of `x'` must be at least one less than the upper bound of `y'` in order to make the equality true. `x'` also cannot exceed its own upper bound. Similarly, the lower bound of `y'` is the maximum of the lower bound of `x'+1` and the lower bound of `y'` before the statement.

The equality test to an untainted variable (Rule 7) will set both bounds to `y` if the tainted value is indeed equal to the value. If the tainted value `x'` is not equal to `y`, there is no change unless `y` happens to equal one of the bounds. In this case, the bound is adjusted accordingly. While in this case it would be possible to split the interval, this is not necessary as only the lower and upper bounds are needed to validate array accesses. In Rule 8, two tainted variables are compared for equality. If they are equal, each variable will have an identical new range that is formed and by taking the highest lower bound and lowest upper bound. If they are not equal, no change is made[1].

The effect of a `while` loop comparison is shown in Rule 9. When the body of the loop is entered, the condition `x' < y` is true and the bounds are updated appropriately. Upon exiting the loop, the bounds are updated to reflect that the condition is now false. `for` loops and `do` loops are handled in the same manner except that the bounds are not updated during the first pass in a `do` loop since the condition is not tested until the end of the loop. The case where two tainted values are compared as a loop condition is analogous to the `if` statement.

Notable omissions from the list are the logical-or (`||`) and logical-and (`&&`) operators. A simplification phase, discussed in Section 3, converts these short-circuited operators into the appropriate if-then-else constructs.

To perform interval bounds checks, it is necessary to keep track of the sizes of all arrays in the program. The size of globally and locally declared arrays are known at compile-time and are straightforward to process. Dynamic variables pose an interesting challenge in our target language C. Since all dynamic memory allocations are considered to be untyped, we consider all dynamic allocations to be a single array with a size equal to that of the memory allocation.

1. There are some cases that we ignore where the bounds could be adjusted. One example would be when the equality is false and `x'.lb == x.'ub == y'.ub`. In this case, `y'.ub` should be lowered by one. Such cases can easily be added if they result in false alarms.

2.1.1 Array reference example

To illustrate our approach, we will describe a bug that was discovered in OpenSSH. It occurred in the channel code (*channel.c*). The relevant code is shown in Figure 6. In function `channels_new`, the channels array is a dynamically growing array with a size equal to `channels_alloc`. The starting size of the array is 10.

Some time after `channel_new` is called, the function `channel_input_data` is invoked. At line 36, an integer is obtained from a packet using `packet_get_int` (an OpenSSH function that grabs the next integer from the current network packet). Upon return of `packet_get_int`, `id` will be tainted with a lower bound that is equal to $-\infty$ and an upper bound equal to ∞. The value of `id` is passed into the function `channel_lookup`, so the parameter `id` upon entry will have the same bounds.

At line 46, there is a check to make sure that index is within the bounds of the array. If the run-time value of `id` is out of the range $0 \leq$ `id` $\leq$ `channels_alloc`, the error would not be detected since the function call returns before the array access. At the array access in line 50, the interval constraint of `id` has a lower bound of zero and an upper bound equal to the run-time value of `channels_alloc`. The `channels` array has a run-time size equal to `channels_alloc`, indexed from zero to `channels_alloc-1`. Since the upper bound of `id` is `channels_alloc`, it exceeds the bounds of the array and an error is declared. If `id` is in the range $0 \leq$ `id` $\leq$ `channels_alloc`, the error will be detected despite the fact that an error only occurs when `id` equal to `channels_alloc`. For any value of `id` that executes the array access, our technique will detect the error. To fix this bug, line 46 must be changed to use '>='.

2.2 Detecting misuse of string functions

String functions such as `strcpy` can lead to software faults since no check is performed to determine if the source string will fit in the destination buffer. In order to check these functions, strings that come from user input are tracked. Since a string provided by a user can have arbitrary length, our technique assumes that all input strings can have infinite length initially. Comparisons made to the length of the string adjust the maximum length of the string. When a string copying function is called, the maximum length of the string is checked to ensure it will fit in the destination.

Another common problem is manipulating strings that are not terminated with a null character. While input strings automatically may contain a null character when they are first created, they could be copied using functions such as `strncpy` which do not copy the null character in all cases. Another common mistake is to forget that the `strlen` command does not include the null character in its count, lead-

```
/* Pointer to an array containing all allocated channels. The array is
 * dynamically extended as needed. */
static Channel **channels = NULL;

/* Size of the channel array. */
static int channels_alloc = 0;

Channel *
channel_new(...)
{
  int i, found;
  Channel *c;

  /* Do initial allocation if this is the first call. */
  if (channels_alloc == 0) {
    channels_alloc = 10;
    channels = malloc(channels_alloc * sizeof(Channel *));
    ...
  }
  ...
  if (found == -1) {
    channels_alloc += 10;
    channels = realloc(channels, channels_alloc * sizeof(Channel *));
    ...
  }
  ...
}

void
channel_input_data(int type, int plen, void *ctxt)
{
  int id;
  Channel *c;

  /* Get the channel number and verify it. */
  id = packet_get_int();                                  -∞ ≤ id ≤ ∞
  c = channel_lookup(id);                                 -∞ ≤ id ≤ ∞
  ...
}

Channel *
channel_lookup(int id)                                    -∞ ≤ id ≤ ∞
{
  Channel *c;

  if (id < 0 || id > channels_alloc) {                    -∞ ≤ id ≤ ∞
    log("channel_lookup: %d: bad id", id);
    return NULL;
  }
  c = channels[id];                                       0 ≤ id ≤ channels_alloc
  return c;
}
```

Figure 6: OpenSSH channel bug. The `channels` array has a size of `channels_alloc`. The bug occurs in `channel_lookup` where it is possible to access `channels[channels_alloc]` which is outside the bounds of the array.

ing to an off-by-one error if used incorrectly. Since we consider input strings to have an arbitrary length, it is not necessary for a user to supply a string of the precise length in order to find such errors.

All strings and arrays in the program are tracked with the three fields: `actual_size`, `max_str_size`, and `known_null`. The field `actual_size` stores the actual run-time size of the array and cannot change (except for calls to `realloc`). The field `max_str_size` stores the maximum size of the string in the array. It refers to the largest possible size of a string that a user can supply. For example, strings that come from the command line have an initial `max_str_size` of infinity (`INT_MAX`) and strings that are created using `fgets` have a `max_str_size` equal to the supplied limit. For arrays that do not contain strings, `max_str_size` is equal to `actual_size`. The `known_null` field is a flag that is true if the string is known

Table 2: Representative Rules for String Buffer Overflow Checking. `s` is an array, `p` is a pointer, `n` and `c` are integers, and `m''` and `n''` are integers that store a string length.

Array Creation		
1	`s = argv[i]`	`s.actual_size = strlen(s)+1; s.max_str_size = INT_MAX; s.known_null = TRUE;`
2	`s: string constant`	`s.actual_size = s.max_str_size = strlen(s)+1; s.known_null = TRUE;`
3	`char s[n]`	`s.actual_size = s.max_str_size = n; s.known_null = FALSE;`
4	`s = malloc(n)`	`s.actual_size = s.max_str_size = n; s.known_null = FALSE;`
5	`s = malloc(n'')`	`s.actual_size = n'';` `s.max_str_size = (n''.string).max_str_size + n''.size_diff;` `s.known_null = FALSE;`
String Length Manipulation		
6	`n'' = strlen(s)`	`Assert: s.known_null == TRUE` `n''.string = s; n''.size_diff = -1;`
7	`n'' = m'' + 1`	`n''.string = m''.string; n''.size_diff = m''.size_diff + 1;`
8	`if (n'' <= c)`	`if (n'' <= c) (n''.string).max_str_size =` `MIN((n''.string).max_str_size, c + n''.size_diff);` NOTE: no change is necessary if `(n'' > c)`
Basic Array Operations		
9	`s[n] = c`	`if (c == 0) s.known_null = TRUE;`
10	`*p = c`	`if (c == 0) (p.array_base).known_null = TRUE;`

to contain a null. If it is false, it is not known if a null character is present or not present. During checking, we assume that the string is not null terminated if `known_null` is false. We will represent accesses to these fields using structure notation: `s.max_str_size` refers to the field `max_str_size` associated with the array `s`.

Strings can be created in a variety of ways as shown in the first five rules of Table 2. Strings that come from the command line or environment variables (Rule 1 in Table 2) will be marked as having an infinite maximum string size since the user could have supplied a string of any length[1]. Since these strings are automatically null-terminated, the `known_null` field is set to true. String constants (Rule 2) are not dependent on the user and thus have a maximum string size equal to its actual size. Rules 3-5 assume the arrays are uninitialized, making the initial value of the `known_null` flag false. Initializers can be viewed as assignments after the array has been created. Locally or globally declared arrays (Rule 3), do not store strings when they are first created and have a maximum string size equal to its actual size. In most cases, dynamically allocated arrays are processed identically to the creation of arrays declared at compile-time (Rule 4). One exception is when the size of the allocation is dependent on the size of another string. This case (Rule 5) is described in the next paragraph.

1. We conservatively use infinite size even though the operating system imposes a limit on the length of a command line.

Similar rules exist for `calloc`, except that `known_null` is initialized to true.

In order to properly adjust the maximum size of a string, it is necessary to track integers that store string lengths. Any integer that is storing a string length will have state that stores the starting address of the corresponding string (denoted using the field `string`). In addition, a `size_diff` field is also stored that is the difference between the value stored in the integer and the actual length of the string. This is important as our approach includes the null character in the length of a string while the `strlen` command does not. Therefore, the initial size difference for a `strlen` result is -1. In Table 2, variables that store string lengths are represented with two tick marks (such as `n''`). Rule 6 shows the `strlen` call. Since `strlen` requires the input string to be null terminated, a check is made to make sure that is the case. Addition and subtraction operations (Rule 7) on the string length adjust the size difference appropriately. For example, adding one to a `strlen` result to account for the null character will result in a size difference of zero. The maximum length of the string can be reduced with a control operation; this is illustrated in Rule 8. If `(n'' <= c)` is true, the maximum size of the string s is adjusted to `c + n''.size_diff` unless the maximum size is already smaller. If `(n'' <= c)` is false, no adjustment is made to `s` since there is no restriction on the maximum size of `s`. Refer back to Rule 5 where a string length is used as the size parameter to a dynamically allocated array. This is

Table 3: Representative Rules for String Buffer Overflow Checking (continued). `s`, `d`, `set`, `fmt` are strings, and `p` is a pointer to a string. `n` is an integer and refers to a parameter that restricts the number of a characters written into a destination buffer. The macro `SIZE(s)` is equal to `MAX(s.actual_size, s.max_str_size)`.

String Functions		
11	strcpy(d,s)	Assert: s.known_null == TRUE Assert: s.max_str_size <= SIZE(d) d.max_str_size = s.max_str_size; d.known_null = TRUE;
12	strncpy(d,s,n)	Assert: s.known_null == TRUE Assert: (n <= SIZE(d)) d.max_str_size = MIN(s.max_str_size, n); d.known_null = (s.max_str_size <= n);
13	strcat(d,s)	Assert: s.known_null == TRUE && d.known_null == TRUE Assert: s.max_str_size <= SIZE(d) - strlen(d) d.max_str_size = s.max_str_size + strlen(d); d.known_null = TRUE;
14	strncat(d,s,n)	Assert: s.known_null == TRUE && d.known_null == TRUE temp_src_size = MIN(n + 1, s.max_str_size) Assert: temp_src_size <= SIZE(d) - strlen(d) d.max_str_size = temp_str_size + strlen(d); d.known_null = TRUE;
15	strchr(p,s) also: strrchr	Assert: s.known_null == TRUE if (p) p.array_base = s;
16	strstr(p,s,set) also: strpbrk, strppbrk, strtok, strsep	Assert: s.known_null == TRUE && set.known_null = TRUE if (p) p.array_base = s;
17	d = strdup(s)	Assert: s.known_null == TRUE d.actual_size = d.max_str_size = s.max_str_size; d.known_null = TRUE;
18	fgets(d, n, stream)	Assert: n <= SIZE(d) d.max_str_size = n; d.known_null = TRUE;
19	gets(d)	Automatic error! d.known_null = TRUE;
20	scanf(fmt, d) Also: fscanf, sscanf	Get width from fmt (width = 0 if no width was given) Assert: width != 0 && width <= SIZE(d) if (width != 0) d.max_str_size = width; d.known_null = TRUE;
21	sprintf(d, fmt, s)	Assert: s.known_null == TRUE Check to make sure the sum of all source strings does not exceed SIZE(d), non strings are ignored in this calculation d.known_null = TRUE;
22	snprintf(d, n, fmt, s)	Assert: s.known_null == TRUE Assert: n <= SIZE(d) If the sum of all source strings exceeds SIZE(d), then d.known_null = FALSE; otherwise d.known_null = TRUE; d.max_str_size = n;
23	strcmp, strpos, strrpos, strspn, strcspn, atof, atoi, atol, strtod, strtol, strtoul, strcoll	Check that all input source strings are null terminated.

commonly done before a string copy to ensure the destination as enough space to hold the source string. As a result, the field `max_str_size` of the newly allocated region is initialized to properly reflect the maximum size of `n''.string` rather than using `n''`. Our current implementation limits the usage of string length integers. We do not handle arithmetic operations except addition and subtraction and we do not handle operations that involve two string length integers. For more information on these restrictions and the limitations they cause, see Section 3.1.

Assigning zero to an element of an array will set the `known_null` flag to true (Rule 9). An assignment of a value other than zero has no effect on the state of the array. The `known_null` flag is also set to true when functions `bzero` and `memset` (to zero) are called. String arrays are often accessed via pointers. This case is handled by shadowing the pointer with the base address of the array (denoted using `array_base`). Pointers that do not point to arrays are not shadowed. In the event, the string is addresses using an interior pointer, the state of the array is obtained using the shadowed base address. This is illustrated in Rule 10. For brevity, we will omit this level of indirection and assume arrays are used in all future rules.

Table 3 shows how string library functions are handled. Rules 11 and 12 illustrate how string copies are handled. In `strcpy`, the source must be null terminated and the size of the maximum string size must fit in the destination. The size of the destination is determined by taking the maximum of `actual_size` and `max_str_size`. In the case where `max_str_size` is larger than `actual_size`, `max_str_size` is chosen since the only way this situation can occur is when the array was dynamically allocated with a string length. Our approach naively assumes that the string length used referred to the length of the source string. This assumption could lead to undetected bugs and is discussed in more detail in Section 3.1. For brevity, we use `SIZE(s) = MAX(s.actual_size, s.max_str_size)` to represent the size of the destination buffers. If both the null check and size check pass, the destination will then have `known_null` set to true and a `max_str_size` equal to that of the source. In `strncpy`, a check ensures that the destination size is less than the supplied size (`n`) parameter. This is done regardless of the size of the source string since nulls are padded at the end if the source is smaller. The destination will have a `max_str_size` that is the smaller of `n` and `max_str_size` of the source. The `known_null` is only true if the entire source string is copied.

The `strcat` functions (Rules 13 and 14) are handled similarly to `strcpy`. The key differences are the destination must be null terminated and the run-time size of the destination string is subtracted from the size comparisons. Unlike `strncpy`, `strncat` will always add a null character and as a result could copy `n + 1` characters. The substring extraction functions (Rules 15 and 16) check for null terminated input strings. The destination, if not NULL, is a pointer to somewhere in the original source string. As a result, the pointer gets shadowed with the address of the source string. This has the effect of assuming that the substring is identical to the original string, which in the worse case, could be true. Token extraction routines such as `strtok` are handled the same way, each token is the assumes to be the same size of the original source string. The `strdup` function (Rule 17) is straightforward. A new string is created with the exact same characteristics as the source string.

Strings that come from input functions such as `fgets` (Rule 18) will have a maximum size equal to the size that was supplied as the limit to the function and are null terminated. These input functions are checked to ensure that input will fit in the supplied buffer. As a result, the function `gets` (Rule 19), which has no checking, automatically flags an error each time it is used. The `scanf` family of functions (Rule 20) are also unsafe unless a field width is supplied for each input string. If a field width is present, it will checked to ensure the input string fits in the destination buffer. The function `sprintf` (Rule 21) is implemented to ensure that the sum of the maximum sizes of all source strings does not exceed the destination size. In `snprintf` (Rule 22), the size is limited by the size parameter but null is not written if the source strings can exceed the destination. String functions that only read strings only check to see if all input strings are properly null terminated (Rule 23).

A programmer may mimic the behavior of a `strcpy` by using a pointer to walk the elements of an array and copying each element individually. Copying via indirect references (`*d = *s`, for example) does not alter state unless the last element of the array `s` is copied into `d`. In this case, we assume that the entire array is copied from `s` into `d` and the statement is treated like a string copy. While this is not the case in all situations, current tool limitations prohibit more sophisticated analysis. This is addressed as future work.

2.2.1 String example

A detailed example illustrating how string errors can be found is shown in Figure 7. The two buffers `buf0` and `buf2` have an initial maximum size equal to their static sizes. In line 6, the input value `argv[1]` is copied into `buf0` using `strncpy`. While the specified size of 12 does not cause an overflow, the null flag for `buf0` remains off since a null would not have been copied if `argv[1]` has at least 12 characters. The `strdup` in line 7 will duplicate the state values of `argv[2]` into `buf1`. If `value` is true, execution will continue to line 10 where the pointer `p` is assigned to point to the second element of `buf0`. This causes `p` to be shadowed with the base address of `buf0`. When `p` is used in the `strcpy`, an error gets properly signalled because `buf0` may not be null terminated. In the case where `value` is false, a comparison is made based on the length of `buf1`. Assuming it is less than or equal to 6, control will be taken to line 14 and the maximum size of `buf1` will be restricted to 7 (6 plus 1 for the null character that `strlen` does not count). The `known_null` flag is set for `buf0` in line 14. In line 15, an error results because the sum of the maximum sizes of the two source buffers (19) can exceed the size of the destination (18).

2.3 Other improper uses of input data

Our approach can also be used detect other situations where

	`char buf0[12];`	buf0.max_str_size = 12, buf0.known_null = FALSE
	`char *buf1;`	
	`char buf2[18];`	buf2.max_str_size = 18, buf2.known_null = FALSE
	`char *p;`	
5		
	`strncpy(buf0, argv[1], 12);`	buf0.max_str_size = 12, buf0.known_null = FALSE
	`buf1 = strdup(argv[2]);`	buf1.max_str_size = ∞, buf1.known_null = TRUE
	`if (value) {`	
10	`  p = buf0 + 1;`	p.array_base = buf0
	`  strcpy(buf2, p);`	p.array_base is buf0 → buf0.known_null == FALSE → ERROR
	`}`	
	`else if (strlen(buf1) <= 6){`	buf1.max_str_size = 7, buf1.known_null = TRUE
	`  buf0[12] = 0;`	buf0.max_str_size = 12, buf0.known_null = TRUE
15	`  sprintf(buf2, "%s%s", buf0, buf1);`	(buf0.max_str_size + buf1.max_str_size = 19) > (buf2.max_str_size = 18) → ERROR
16	`}`	

Figure 7: Example of detecting string bugs. The `strcpy` in line 11 can fail because `buf0` is not null terminated. The `sprintf` in line 15 can fail because the sizes of the two source strings could exceed the size of the destination.

`unsigned int nresp;`	
`nresp = packet_get_int();`	$0 \leq$ nresp $\leq \infty$
`if (nresp > 0) {`	
`  response = malloc(nresp * sizeof(char*));`	$1 \leq$ nresp $\leq \infty$
`  for (i = 0; i < nresp; i++)`	$1 \leq$ nresp $\leq \infty$
`    response[i] = packet_get_string(NULL);`	
`}`	

Figure 8: OpenSSH challenge bug. Unbounded data from a packet can cause overflow when calling `malloc`.

input data could be used dangerously and possibly lead to software faults. Unconstrained input used to control the number of loop iterations, the size of a memory copy, or the size of a memory allocation could be dangerous [1]. We check to make sure that the variables controlling these uses have been constrained in some fashion. An error is signalled if the upper bound is equal to the maximum value allowed by the type of the variable. An error is also reported if the value could be negative. For these situations, it is important to note that these types of uses are not always errors. In some cases, input is constrained later within the loop and malicious behavior is properly thwarted. In other cases, the loop may not do anything that can be exploited. Memory allocations are likely not dangerous if the output is properly checked to ensure the allocation was successful.

Another related problem is arithmetic overflow. A common example of the case of adding two large signed integers where the destination is not large enough to store the result, leading to a negative number with large magnitude. As with the previous case, this usually occurs when input data is unconstrained. Overflow and underflow is detected using the bounds associated with integers. Once an operation is completed, the resulting upper and lower bounds are analyzed to determine if it can fit into the destination variable. An error is signalled if the resulting value cannot fit.

While this type of problem doesn't necessarily lead to an security exploit, it often does. For example, we describe another security bug found in OpenSSH. The code is listed in Figure 8. Data is received from a packet and then is subsequently used to allocate an array. At the time of the `malloc`, no restriction has been placed on the input. A malicious user could supply an extremely large value in order to cause an overflow on the multiplication in the `malloc` call resulting in a small allocation. Since the same input value controls the number of iterations within the loop that follows, the array accesses within the loop can be used to access memory outside of the array.

Our approach can also be used to find potential bugs when integers are casted. For example, an assignment of a signed long integer into an unsigned short integer can be problematic if the signed long integer has a negative value or a value larger than the maximum size of the unsigned short integer. However, during our testing, we were unable to find any defects due to improper casting. We did find several cases where casting of this sort was done intentionally and correctly causing false alarms. As a result, we disabled casting checking for our experiments in Section 4.

3. Implementation

Our dynamic checker is built on a general purpose source-level instrumentation tool, called *MUSE*, that we designed to facilitate the construction of dynamic defect detection tools. Unlike most dynamic verifiers that focus on one particular property, our system is general purpose. A user, using our checking specification language, can specify program properties they wish to validate.

MUSE is a source-level general purpose program instrumentation tool, allowing users to specify the properties they are interested in checking. The tool is built as a compilation phase in the GNU GCC compiler. Given a hand-written model of program correctness, MUSE will automatically

locate the instrumentation points used by the model, graft in the necessary program instrumentation, and link in any needed additional run-time support. No modifications to the program source code are needed. When the instrumented program is executed, any violations of the correctness properties specified are detected and reported to the user. Program instrumentation is performed at the abstract syntax tree (AST) level, thus source code is required to add instrumentation. Functions without source code, such as system libraries, m ay be instrumented at their entry and exit points.

The first step of the instrumentation process is to simplify the program. We convert the program into *Elemental C*, an intermediate C representation similar to the simple grammar developed by Hendren *et al.* [15]. The purpose of simplification is to reduce the complexity of identifying and instrumenting relevant program points. Complex C statements are broken down into simple statements with at most two operands and a single assignment to an l-value (such as a = b + c). Side effects and short-circuited operators are eliminated via program transformations.

The MUSE correctness specification consists of program source patterns and associated model actions that are specified as a collection of <pattern, action> tuples. Program source patterns are simple regular expressions, including wildcards, that are matched against elemental C statements. The actions are completely written in C and contain the instrumentation functions that perform state management and error checking. Actions are compiled with the instrumented program to form an instrumented executable. The patterns correspond to complete statements or subexpressions within the elemental C language. In addition, patterns exist that match special program events. Examples of special events include the beginning of a function or use of a variable as an r-value. It is beyond the scope of this paper to present the specification language.

During the instrumentation phase, the patterns are parsed and the program is traversed one elemental C statement at a time. When there is match, the action code is inserted at the matching site. Instrumentation can either be added before or after the matching site depending on how the instrumentation was specified. In addition, various run-time values may be passed to the instrumentation in order to parameterize model actions. After instrumenting the program AST, the remaining compiler phases are executed to produce an instrumented executable. Compiler optimizations may be enabled during this portion of the compilation. Optimizations can recoup some of the performance penalty that is incurred by instrumentation.

The model is implemented using three tables that contain the shadowed state associated with the variables. One table stores state for arrays, another for pointers, and one for integers. All arrays are inserted in the array table when they are created and are indexed by their base address. Each entry in the array table contains four fields: `actual_size`, `max_str_size`, `known_null`, and `is_input`. The first three fields are described in Section 2.2. The `actual_size` field is also used in the array reference checking. The `is_input` field is used to mark arrays that contain program input. If an array from input is used in a function that converts a string to an integer, such as `atoi`, the resulting integer will be treated as input. The pointer table only stores pointers that refer to an array and are indexed by the address of the variable. Each entry contains a single field, the base address of the array that the pointer points to. The integer table also only contains entries for integers that require shadowed state. An integer can require state for three reasons: (a) it contains input data, (b) it contains string length data, or (c) it is a boolean value that will narrow the bounds if used in a conditional expression. A flag is used to distinguish between the three cases. In case (a), there are upper and lower bound fields as described in Section 2.1. In case (b), there are string and size difference fields as in Section 2.2. In case (c), the entry contain an address indicating the variable to be updated, the appropriate bounds (`lb`, `ub`, and `max_str_size`) when the condition is true and bounds for when the condition is false. If the conditional value is used in a control statement, the bounds of each variable is updated using this information.

When an error is detected, the error message will displayed that includes the file name, line number, matching MUSE pattern, and a descriptive message describing the error. The error is detected at the point of the dangerous use such as an array reference or string function. However, the source of the error may not be near the dangerous use. With the help of a debug mode, that prints out a message every time state has changed, it was usually very straightforward to find the source of the error or to classify the bug as a false alarm.

3.1 Limitations

Since our approach is dynamic and relies on the particular control path taken through a program, it is an ***unsound*** approach, meaning that is possible to miss the detection of actual bugs. With respect to a particular control path, our approach is also unsound. One problem stems from the use of run-time data. An example is detecting when a zero gets written into an array (see Rules 9 and 10 in Table 2). Another case is the actual size of the array is used during array checks. On a different run with the same control path, the size of the array, if controlled by input, could be smaller and be subjected to an array buffer overflow. This is illustrated in Figure 9. The size of the array is controlled by user input and could be any value from 1 to 10. However, the filter of illegal accesses for the index in line 13 is valid when the array size if 10 and invalid for all other accesses. As a result, an error will be missed if 10 is supplied as the array size.

```
unsigned int size;
unsigned int index;
int *array;
int x;

size = getchar();
if (size <= 0 || size > 10) exit();
array = (int *) calloc(size, sizeof(int));

/* initialize array */

index = getchar();
if (index < 0 || index > 9) exit();
y = array[index];
```

Figure 9: Example of an unsound control path. This code segment will overflow the array if x is 4.

Fully addressing these problems would require symbolic analysis. Other shortcomings occur due to a lack of symbolic analysis. While most result in false alarms, a case that results in a missed bug is when the string length of is used to allocate an array and a subsequent `strcpy` operation copies an entirely different string into the destination. This is a bug if the size of the second string is larger than the first. If the `max_str_size` of the first string is high, an error will likely be missed. However, using `actual_size` results in too many false alarms. Lastly, we are also unsound in that we do not attempt to catch every type of buffer overflow that is possible.

Our technique is also *incomplete* in that can produce false alarms, signalled bugs that are not actually bugs. As eluded to earlier, these often occur due to a lack of symbolic analysis. Not all possible relationships between different strings or variables are tracked and this can cause operations that narrow bounds to be missed. A specific example of where symbolic analysis is not present is the limited functionality associated with integers that store string lengths. When an unsupported operation occurs, a warning is emitted, and the result is no shadowed. Another problem arises in that our technique does not keep track of which position the null character is in. In order to maximize the number of detected bugs, we assume it is in the last position. This assumption also leads to an increase in false alarms. In practice, we found the number of false alarms to be manageable. We describe the false alarms triggered in Section 4.1.

4. Results

Our dynamic input analysis checker has been applied to the eight programs listed in Table 4. Programs were compiled using GNU GCC with an -O4 optimization level. Using these programs, we sought to find bugs and measure the effect our instrumentation had on run-time performance.

Three of the programs (*anagram*, *ks*, and *yacr2*) are from the pointer-intensive benchmark suite [26] and were selected due to difficulty of analyzing these programs statically. Each benchmark included several test inputs. All inputs were run for testing purposes. For performance testing, the largest input was selected.

Table 4: Programs used during testing

Program	Description of Program	Defects Found	False Alarms
anagram	anagram generator	2	0
betaftpd	file transfer protocol daemon	1	1
gaim	instant messenger	1	1
ghttpd	web server	3	2
ks	graph partitioning	4	0
openssh	openssh secure shell	3	1
thttpd	web server	0	1
yacr2	yet another channel router	2	1

The other five programs are networking applications. The popular secure shell program *openssh* was tested by targeting the two known bugs that were discovered. Additional testing focused on different modes in both the server and client. Performance testing was done used a scripted session that involved transferring large files and does not attempt to exhaustively execute all of the code. We also tested *gaim*, a popular instant messaging program. Testing was purely interactive and several different instant messaging protocols (MSN, Yahoo, etc.) were used. The one bug found in *gaim* was in the initialization code and was independent of the protocol used. Due to the interactive nature of *gaim*, it was not used in the performance experiment. For the FTP and web servers, testing was done by having the server process several FTP and HTTP requests in different configurations. Performance testing was done by a script that consisted of several requests for a file or web page.

In our testing, we did not strive to exhaustively test all possible code within a program. Using MUSE, the user can add a coverage mechanism similar to one in [14]. The coverage technique may be used to make sure that all potentially dangerous statements (array references, pointer dereferences, and string functions) are executed at least once. Like normal statement coverage, executing all of the dangerous statements once does not guarantee that all bugs will be found since bugs could be dependent on the particular control path that is executed.

4.1 Bugs Detected

With our tool, we were able to find 16 bugs, shown in Table 4. In order to compare our results with static analysis, we used the tool from [29] and analyzed six of the eight programs. Problems processing the source code prohibited analysis of *gaim* and *openssh*. The tool was unable to detect any of the bugs that were discovered using our approach and did not detect any additional bugs.

Two of the three defects found in *openssh* (described in Section 2) are both security flaws present in version 3.0.2. The channel id bug would be difficult to locate via static analysis. The array is dynamically allocated and its size can change during execution. In addition, creation of the array, reading of input, and accessing the channel array each occur in three distinct functions. Any static approach to locating this bug would require interprocedural analysis. The third defect discovered in *openssh* is an addition overflow problem where two numbers read from network data are added together. While this bug does not create a security exploit or a crashing program, it could lead to unexpected program behavior.

In *gaim*, a defect occurs when reading the configuration file. Each field is placed into a large temporary buffer. The fields are processed and copied into the appropriate data structure. In some cases, the fields are copied into a smaller buffer without checking to see if it will properly fit. Examples of fields where this occurs are the username and password. While this bug could not be exploited remotely, it could cause the program to crash. The three defects in *ghttpd* were all due to misuse of string functions. In one case, a `strncat` function contains a limit that does not account for the null character. For a given limit `n`, it is possible for `n + 1` characters to be written since a null character is always written. Another defect was caused by calling strstr on an uninitialized local array. The third defect in *ghttpd* and the defect found in *betaftpd* were the result of using data received from the network without any guarantee that there is a null character.

One bug in *anagram* permits a user to overflow a buffer with characters from an input file. The buffer is dynamically allocated in proportion to the size of the input file. Extra space is added to store additional information about each word in the file. The size of the extra space is controlled by a fixed compile-time constant representing the maximum number of words allowed in the file. If the file contains more words than this constant, the buffer could overflow. The other bug is the result of using `gets`, automatically a dangerous function. In *ks*, two bugs resulted from an input being used to reference an array without any checking to see if it exceeded the array bounds. The other two defects were due to undetected arithmetic overflow with an input value and a loop based on input that is not checked. Both defects discovered in *yacr2* were due to a multiplication overflow. These bugs are very similar to the OpenSSH challenge bug in Figure 8.

Another important factor in bug detection systems is the number of false alarms - situations where an error is signalled when no defect occurs. During the course of our testing, we detected seven false alarms. Three of the seven cases (*betaftpd*, *gaim*, *ghttpd*) were situations where a loop controlled by input did not result in a bug. The other false alarm in *ghttpd* was due to `sprintf` function that was used to concatenate two strings into a new string. The destination buffer had a size equal to the combined sizes of the two strings. This is a case where lack of string length support for the addition of two string lengths leads to a false alarm.

In *openssh*, an arithmetic overflow bug was detected but a consistency check after the operation would correctly signal a failure. In *thttpd*, a false alarm occurs because our approach conservatively assumes that the null character is in the last possible position and does not track the precise location of the null character within an array. A buffer overflow is signalled incorrectly because the program guarantees that a null is in the first position of an array. The false alarm in *yacr2* is due to reading the input file twice. It is read once to set the array sizes and a second time to initialize the array values. Since the array sizes were based on the input, no errors can occur. Our tool currently has no mechanism for determining that the input is actually constrained when it is read the second time.

4.2 Performance

In order to test run-time performance, all of the programs were run on a lightly loaded 1.8 GHz Pentium IV computer running Linux. Program run-times measured using the Unix `time` command were compared to an uninstrumented program running the same test input. The results of the experiment are shown in Table 5. It is important to point out that we have not focused any effort on performance optimization at this point. The results show that there is a large opportunity for improvement possible. The amount of slow-down experienced is dependent on the program. The five server programs exhibited the least amount of slow down with *betaftpd* having the least with 13x. The three pointer intensive benchmarks suffered significant slow-down from a factor of 162x in *anagram* to 220x in *ks*. The disparity in the results can be attributed to the fact that the pointer intensive benchmarks have more integer processing than the servers.

The breakdown of the dynamic instrumentation sites is shown in Table 6. *Array state* sites keep track of array information such as dynamic array sizes and state associated with strings. *Array references* sites include a call to an array reference check function when the index has been controlled by input. *Pointer manipulation* sites are calls to track pointers with their associated array. *Integer state* sites call an associated function to propagate and adjust interval constraints and string lengths. *Control points* are calls that narrow interval constraints or maximum string lengths. *String functions* sites include calls to check the input strings. *Other* includes miscellaneous instrumentation that does not fit the earlier categories. The *Useless* column refers to instrumentation that did not manipulate any of the instrumented states. This includes 1) integer operations and

Table 5: Run-time performance. Performance slow-down is large for computation heavy programs such as *anagram* and *yacr2*. The slow-down is considerably less for server programs *openssh* and *thttpd*.

Program	Simple Stmts	Code Size (Kb)			Run Time (seconds)			Static Sites	Dynamic Sites	Heap Array Refs
		Orig	New	Increase	Orig	New	Increase			
anagram	1,274	11	190	16.8	0.11	17.79	162	9,20	69,881,404	10.3%
betaftpd	6,325	31	657	21.1	0.08	1.09	13	5,363	3,768,054	69.2%
gaim[1]	374,623	1620	35500	21.9	N/A	N/A	N/A	262,303	N/A	11.0%
ghttpd	3,755	27	424	15.6	0.34	6.70	20	3,971	97,017,000	0.0%
ks	2,066	13	237	18.0	8.75	1923.62	220	2,086	1,889,043,968	0.0%
openssh	155,835	885	9235	10.4	0.02	0.38	19	105,900	421,551	1.8%
thttpd	22,758	138	2301	16.7	0.32	8.47	26	15,530	29,210,392	50.0%
yacr2	7,999	33	786	23.8	0.55	96.79	176	6,454	392,839,706	100.0%

1. Due to the interactive nature of gaim, we were unable to accurately measure its run time performance.

Table 6: Breakdown of dynamic instrumentation calls.

Program	Array State	Array References	Pointer Manipulation	Integer State	Control Points	String Functions	Other	Useless
anagram	2.7%	0.9%	15.0%	3.7%	1.3%	0.0%	2.8%	73.7%
betaftpd	4.4%	0.0%	10.1%	0.0%	0.0%	0.0%	4.3%	81.2%
gaim	2.7%	0.0%	13.2%	1.9%	0.4%	0.4%	4.7%	77.3%
ghttpd	0.8%	0.0%	1.6%	0.0%	0.0%	0.3%	0.5%	96.7%
ks	0.0%	0.7%	34.2%	7.5%	6.3%	0.0%	1.1%	50.1%
openssh	2.1%	0.0%	2.6%	5.7%	3.6%	2.1%	5.1%	78.8%
thttpd	2.4%	0.0%	14.3%	3.1%	1.4%	0.2%	0.8%	77.8%
yacr2	0.7%	7.6%	3.3%	11.3%	1.4%	0.0%	1.2%	75.2%

control points that did not manipulate input data or string lengths, 2) pointer operations that were not associated with an array, and 3) array references that did not have an index that was controlled by input. Clearly, a high percentage of instrumentation calls perform no useful action. Many of these instrumentation sites could likely be eliminated by introducing instrumentation-specific optimizations to the compiler, such as performing copy propagation for interval constraints. Eliminating useless instructions will also reduce the code size overhead. Programs with the fewest number of useless instrumentation exhibited the largest slowdowns. This is to be expected because a useful instrumentation site executes more code than a useless one. The effect on code-size is fairly significant. While the instrumentation functions consume 91KB of space, most of the overhead is due to the added instrumentation calls.

Another interesting statistic is the percentage of array accesses on the heap, shown in the last column of Table 5. Heap array sizes are not generally known at compile-time; thus they are more challenging to analyze statically. The programs *betaftpd*, *thttpd*, and *yacr2* have more heap array references than non-heap array references. In fact, *yacr2* has no non-heap array references. The other programs have significantly fewer non-heap array accesses with *ks* and *ghttpd* having none.

5. Related Work

Several dynamic defect detection tools have been developed. Haugh and Bishop [14] check all of the interesting string library functions by comparing the allocated sizes of the arrays. This approach is similar to our string library maximum size checks. Their tool tracks coverage to ensure that each interesting string function is executed once. Our technique can potentially find more defects because it checks for proper null termination and array references. Examples of memory access checking tools include GNU's checker [6] and Purify [13] which detect memory bugs by keeping track of the state of dynamically allocated memory. Electric Fence [24] places inaccessible pages before and after each dynamically allocated object. A segmentation fault occurs if an access occurs outside the object. CCured [22] uses static program analysis to prove as many pointers to be memory safe as possible. For pointers where this is not possible, run-time checks are inserted into the program. Safe C [2] associates extra state with each pointer that holds the bounds of the object the pointer references. Accesses are compared to the bounds to see if an error occurs. Parasoft's Insure++ [23] checks for a variety of different errors such as memory reference errors, memory leaks, unsafe I/O operations and data conversion errors. The checking is accomplished by adding checking and testing instrumentation around each line of source code. CodeCenter [19] interprets C code and provides run-time type checking and

memory access checking. Fuzz testing [12] found several bugs by injecting a random input stream into Windows and UNIX applications. The work clearly demonstrates that even mature programs pose vulnerabilities to improperly bound inputs.

Several efforts have focused on preventing malicious behavior [8, 20, 21, 25]. The taint mode of Perl [25] can be used to prevent untrusted programs from gaining superuser access. StackGuard [8] is a run-time approach that adds a randomized canary word just below the return address. If the canary word is modified, an error occurs. Buffer overflow attacks that overwrite the return address will also overwrite the canary word negating a jump to the attacker's code. Wahbe *et al.* [30] introduce address sandboxing. An untrusted code module (binary) is instrumented to restrict accesses to that module's segment(s). Related to the issue of security is trust. Proof-carrying code [21] involves embedding a proof into a binary that shows the code satisfies a particular property. An untrusted binary can be verified to see if the proof is valid.

Earlier efforts have used similar bug detection techniques, but the analysis is performed at compile-time using symbolic execution [1, 5, 7, 9, 27, 29]. In these systems, input values are assumed to take on any value and symbolic calculations are used to check if array accesses are within the bounds of the array. Coen-Porisini *et. al.* [7] give a good overview of the approach for a subset of the C programming language and have applied their technique to safety-critical software systems. The advantage of our approach is that our analyses can yield greater precision since they can use run-time information in situations that are difficult to analyze statically. In [1], Ashcraft and Engler constructed models to catch inappropriate uses of tainted data. In their model, tainted data becomes untainted if any check occurs. They only validate that checks are executed. Their approach is unable to determine if the checks are correct. While they have some support for finding errors across functions, their analysis is predominantly local. They have found several bugs in Linux and OpenBSD. PREfix [5] uses a bottom-up approach for model checking. The call graph for the program is created and leaf functions are processed first and replaced with a summary model when called by other functions. Evans and Larochelle [9] and Rugina and Rinard [27] use static analysis to find potential buffer overflows. In [29], arrays or strings are represented as ranges and the problem is transformed into a system of integer range constraints.

Another method for preventing malicious behavior is to enforce safety at the programming language level. Cyclone [18] is a modified form of C that checks pointer accesses using fat pointers. The programmer can limit the number of checks by declaring pointers to be safe. To ensure safety, additional restrictions (for example, disallowing arithmetic) are placed on safe pointers. Shankar *et. al.* [28] introduce a tainted type qualifier to detect vulnerabilities in format strings at compile-time. Data that come from untrusted sources will have a tainted qualified type. When a tainted typed variable is used in potentially dangerous situations, an error is signalled.

A closely related area of study is the elimination of array bounds checks [4, 11]. The approach works by propagating the constraints implied by array bounds checks through the dataflow graph of a program. Similar to our dynamic constraint modifications, program operators and control points adjust propagated constraints accordingly. Array bounds checks that see reaching definitions of earlier (sufficient) checks can be eliminated. These optimizations are directly applicable to the optimization of our input bounds checks, and they form the basis for our on-going static optimization work details in the paper conclusions.

Model checking is another formal method of proving that a program is bug-free. These techniques are very powerful but in order to prove that no errors exist, the number of possible states that must be searched is often extremely large, making the proof infeasible. To reduce the search space, further abstractions are often used that limit the scope of the search. Consequently, the ability to prove that a property is satisfied may be lost, but previous efforts have shown that a large number of bugs can still be found. Static software verification systems include Microsoft's SLAM system [3] which converts a program into a boolean program using predicate abstraction [10]. Reachability analysis is used to determine if an error state can be reached. BLAST [16] uses lazy abstraction, an automated abstraction and refinement process that abstracts the program to the proper amount of precision necessary to verify a particular property. The SPIN model checker [17] is popular for verifying distributed system protocols.

6. Conclusions and Future Work

In this paper, we describe an approach for dynamically checking for software faults caused by improperly bounded program input. Our dynamic approach overcomes many limitations of static analysis while reducing the dependence on the input. On a given program path, the range of all possible input values is validated to ensure that no input-related errors can occur. This is accomplished by shadowing all inputs with state variables.

Integers are shadowed with an interval constraint containing the bounds an input variable may hold. Control decisions narrow the interval constraint and operations adjust the constraints. At potentially dangerous array access sites, the range of variable is checked against the bounds of the referenced array. Strings are shadowed by the maximum possible length the string may hold and a flag that indicated

if the string must be null terminated or not. Control decisions based on the string length can reduce the maximum size. String functions are checked to make sure that source strings are properly null terminated and destination strings have sufficient space for all possible string sizes.

Our approach is generic and can be applied to any program. We applied our technique to eight programs and found a total of 16 bugs including two security flaws in OpenSSH. The cost of our checker's accuracy is run-time performance, with some programs experiencing more than two orders of magnitude slowdown. As such, in its current form our approach is best targeted to development testing where precision is more important.

In the future, we plan to address the performance impacts by adding an analysis phase to the compiler that will eliminate unnecessary instrumentation. This phase will be aware of how the instrumentation works, making possible optimizations such as copy propagation on the input interval constraints. The static analysis can also be used to improve the quality and scope of bug detection. For example, static analysis could identify manual `strcpy` loops that copy elements individually. Another avenue for future work is to address areas of unsoundness by adding symbolic analysis support. This will also reduce the number of false alarms.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments. This work is supported under the DARPA/MARCO Gigascale Silicon Research Center and a National Science Foundation Graduate Fellowship. Equipment support was provided by Intel.

References

[1] K. Ashcraft and D. Engler. Using Programmer-Written Compiler Extensions to Catch Security Holes. Proceedings of the 2002 IEEE Symposium on Security and Privacy, May 2002.

[2] T. Austin, S. Breach, and G. Sohi. Efficient Detection of All Pointer and Array Access Errors. Technical Report #1197, Computer Science Department, University of Wisconsin, Dec. 1993.

[3] T. Ball and S. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. Workshop on Model Checking of Software, May 2001.

[4] R. Bodik, R. Gupta, and V. Sarkar. ABCD: Eliminating Array Bounds Checks on Demand. Proceedings of the Conference on Programming Language Design and Implementation, June 2000.

[5] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. Software Practice and Experience, July 2000.

[6] Checker. http://www.gnu.org/software/checker/checker.html

[7] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezzè. Using Symbolic Execution for Verifying Safety-Critical Systems. Proceedings of the 9th International Symposium on Foundations of Software Engineering, Sept. 2001.

[8] C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. Proceedings of the 7th USENIX Security Conference, Jan. 1998.

[9] D. Evans and D. Larochelle. Improving Security Using Extensible Lightweight Static Analysis. IEEE Software, Jan./Feb. 2002.

[10] C. Flanagan and S. Qadeer. Predicate Abstraction for Software Verification. Proceedings of the Symposium on Principles of Programming Languages, Jan. 2002.

[11] R. Gupta. A Fresh Look at Optimizing Array Bound Checks. Conference on Programming Language Design and Implementation, June 1990.

[12] J. Forrester and B. Miller. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. Proc. of the 4th USENIX Windows System Symposium, Aug. 2000.

[13] R. Hastings and B. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. 1992 Winter USENIX Conference, Jan. 1992.

[14] E. Haugh and M. Bishop. Testing C Programs for Buffer Overflow Vulnerabilities. Proceedings of the 10th Network and Distributed System Security Symposium, Feb. 2003.

[15] L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan. Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations. Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing, Aug. 1992.

[16] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. Proceedings of the Symposium on Principles of Programming Languages, Jan. 2002.

[17] G. Holzmann. The Spin Model Checker. IEEE Transactions on Software Engineering, May 1997.

[18] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A Safe Dialect of C. Proceedings of the USENIX Annual Technical Conference, June 2002.

[19] S. Kaufer, R. Lopez, and S. Pratap. Saber-C: an interpreter-base programming environment for the C language. Proceedings of the Summer USENIX Conference, 1988.

[20] K. Lhee and S. Chapin. Type-Assisted Dynamic Buffer Overflow Detection. Proceedings of the 11th USENIX Security Symposium, Aug. 2002.

[21] G. Necula and P. Lee. Safe Kernel Extensions Without Run-Time Checking. Proceedings of Operating Systems Design and Implementation, Oct. 1996.

[22] G. Necula, S. McPeak, and W. Weimer. CCured: Type-Safe Retrofitting of Legacy Code. Proceedings of the Symposium on Principles of Programming Languages, January 2002.

[23] Parasoft Corporation. Insure++: An Automatic Runtime Error Detection Tool. Technical Report PS961-INS1.

[24] B. Perens. Electric Fence. <http://sunsite.unc.edu/pub/Linux/devel/lang/c/ElectricFence-2.0.5.tar.gz>

[25] Perl v5.6 Documentation: perlsec. <http://www.perldoc.com/perl5.6/pod/perlsec.html>

[26] Pointer-Intensive Benchmark Suite <http://www.cs.wisc.edu/~austin/ptr-dist.html>

[27] R. Rugina and M. Rinard. Symbolic Bounds Analysis of Pointers, Array Indices, and Accesses Memory Regions. Proceedings of the Conference on Programming Languages Design and Implementation, June 2000.

[28] U. Shankar, K. Talwar, J. Foster, and D. Wagner. Detecting Format-String Vulnerabilities with Type Qualifiers. Proceedings of the 10th USENIX Security Symposium, Aug. 2001.

[29] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. Network and Distributed Security Symposium, Feb. 2000.

[30] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. Proceedings of the 14th Symposium on Operating System Principles, June 1993.

Storage-based Intrusion Detection: Watching storage activity for suspicious behavior

Adam G. Pennington, John D. Strunk, John Linwood Griffin,
Craig A.N. Soules, Garth R. Goodson, Gregory R. Ganger
Carnegie Mellon University

Abstract

Storage-based intrusion detection allows storage systems to watch for data modifications characteristic of system intrusions. This enables storage systems to spot several common intruder actions, such as adding backdoors, inserting Trojan horses, and tampering with audit logs. Further, an intrusion detection system (IDS) embedded in a storage device continues to operate even after client systems are compromised. This paper describes a number of specific warning signs visible at the storage interface. Examination of 18 real intrusion tools reveals that most (15) can be detected based on their changes to stored files. We describe and evaluate a prototype storage IDS, embedded in an NFS server, to demonstrate both feasibility and efficiency of storage-based intrusion detection. In particular, both the performance overhead and memory required (152 KB for 4730 rules) are minimal.

1 Introduction

Many intrusion detection systems (IDSs) have been developed over the years [1, 23, 29], with most falling into one of two categories: network-based or host-based. Network IDSs (NIDS) are usually embedded in sniffers or firewalls, scanning traffic to, from, and within a network environment for attack signatures and suspicious traffic [5, 25]. Host-based IDSs (HIDS) are fully or partially embedded within each host's OS. They examine local information (such as system calls [10]) for signs of intrusion or suspicious behavior. Many environments employ multiple IDSs, each watching activity from its own vantage point.

The storage system is another interesting vantage point for intrusion detection. Several common intruder actions [7, p. 218][34, pp. 363–365] are quite visible at the storage interface. Examples include manipulating system utilities (e.g., to add backdoors or Trojan horses), tampering with audit log contents (e.g., to eliminate evidence), and resetting attributes (e.g., to hide changes). By design, a storage server sees all changes to persistent data, allowing it to transparently watch for suspicious changes and issue alerts about the corresponding client systems. Also, like a NIDS, a storage IDS must be compromise-independent of the host OS, meaning that it cannot be disabled by an intruder who only successfully gets past a host's OS-level protection.

This paper motivates and describes storage-based intrusion detection. It presents several kinds of suspicious behavior that can be spotted by a storage IDS. Using sixteen "rootkits" and two worms as examples, we describe how fifteen of them would be exposed rapidly by our storage IDS. (The other three do not modify stored files.) Most of them are exposed by modifying system binaries, adding files to system directories, scrubbing the audit log, or using suspicious file names. Of the fifteen detected, three modify the kernel to hide their presence from host-based detection including FS integrity checkers like Tripwire [18]. In general, compromises cannot hide their changes from the storage device if they wish to persist across reboots; to be re-installed upon reboot, the tools must manipulate stored files.

A storage IDS could be embedded in many kinds of storage systems. The extra processing power and memory space required should be feasible for file servers, disk array controllers, and perhaps augmented disk drives. Most detection rules will also require FS-level understanding of the stored data. Such understanding exists trivially for a file server, and may be explicitly provided to block-based storage devices. This understanding of a file system is analogous to the understanding of application protocols used by a NIDS [27], but with fewer varieties and structural changes over time.

As a concrete example with which to experiment, we have augmented an NFS server with a storage IDS that supports online, rule-based detection of suspicious modifications. This storage IDS supports the detection of four categories of suspicious activities. First, it can detect unexpected changes to important system files and binaries, using a rule-set very similar to Tripwire's. Second, it can detect patterns of changes like non-append modification (e.g., of system log files) and reversing of inode times. Third, it can detect specifically proscribed content changes to critical files (e.g., illegal shells inserted into `/etc/passwd`). Fourth, it can detect the appearance of specific file names (e.g., hidden "dot" files) or content (e.g., known viruses or attack tools). An administrative interface supplies the

detection rules, which are checked during the processing of each NFS request. When a detection rule triggers, the server sends the administrator an alert containing the full pathname of the modified file, the violated rule, and the offending NFS operation. Experiments show that the run-time cost of such intrusion detection is minimal. Further analysis indicates that little memory capacity is needed for reasonable rulesets (e.g., only 152 KB for an example containing 4730 rules).

The remainder of this paper is organized as follows. Section 2 introduces storage-based intrusion detection. Section 3 evaluates the potential of storage-based intrusion detection by examining real intrusion tools. Section 4 discusses storage IDS design issues. Section 5 describes a prototype storage IDS embedded in an NFS server. Section 6 uses this prototype to evaluate the costs of storage-based intrusion detection. Section 7 presents related work. Section 8 summarizes this paper's contributions and discusses continuing work.

2 Storage-based Intrusion Detection

Storage-based intrusion detection enables storage devices to examine the requests they service for suspicious client behavior. Although the world view that a storage server sees is incomplete, two features combine to make it a well-positioned platform for enhancing intrusion detection efforts. First, since storage devices are independent of host OSes, they can continue to look for intrusions after the initial compromise, whereas a host-based IDS can be disabled by the intruder. Second, since most computer systems rely heavily on persistent storage for their operation, many intruder actions will cause storage activity that can be captured and analyzed. This section expands on these two features and identifies limitations of storage-based intrusion detection.

2.1 Threat model and assumptions

Storage IDSs focus on the threat on of an attacker who has compromised a host system in a managed computing environment. By "compromised," we mean that the attacker subverted the host's software system, gaining the ability to run arbitrary software on the host with OS-level privileges. The compromise might have been achieved via technical means (e.g., exploiting buggy software or a loose policy) or non-technical means (e.g., social engineering or bribery). Once the compromise occurs, most administrators wish to detect the intrusion as quickly as possible and terminate it. Intruders, on the other hand, often wish to hide their presence and retain access to the machine.

Unfortunately, once an intruder compromises a machine,

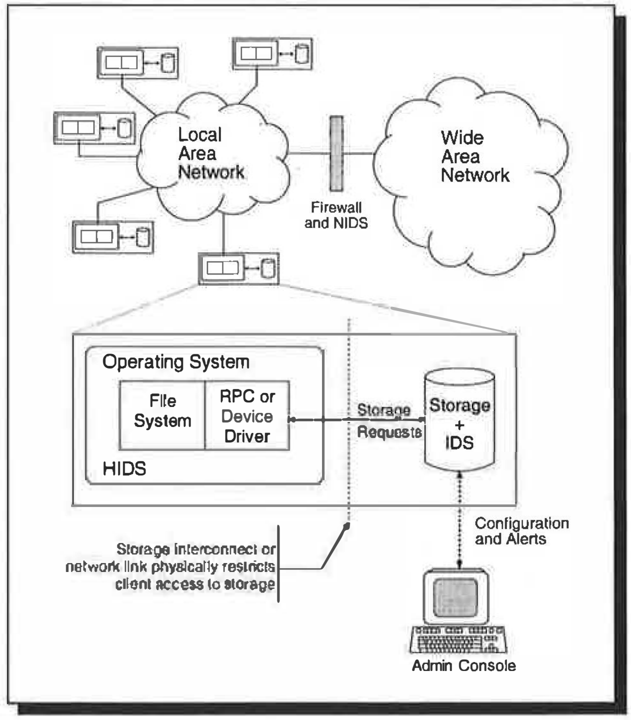


Figure 1: **The compromise independence of a storage IDS.** The storage interface provides a physical boundary behind which a storage server can observe the requests it is asked to service. Note that this same picture works for block protocols, such as SCSI or IDE/ATA, and distributed file system protocols, such as NFS or CIFS. Also note that storage IDSs do not replace existing IDSs, but simply offer an additional vantage point from which to detect intrusions.

intrusion detection with conventional schemes becomes much more difficult. Host-based IDSs can be rendered ineffective by intruder software that disables them or feeds them misinformation, for which many tools exist. Network IDSs can continue to look for suspicious behavior, but are much less likely to find an already successful intruder—most NIDSs look for attacks and intrusion attempts rather than for system usage by an existing intruder [11]. A storage IDS can help by offering a vantage point on a system component that is often manipulated in suspicious ways *after* the intruder compromises the system.

A key characteristic of the described threat model is that the attacker has software control over the host, but does not have physical access to its hardware. We are not specifically trying to address insider attacks, in which the intruder would also have physical access to the hardware and its storage components. Also, for the storage IDS to be effective, we assume that neither the storage device nor the admin console are compromised.

2.2 Compromise independence

A storage IDS will continue watching for suspicious activity even when clients' OSes are compromised. It capital-

izes on the fact that storage devices (whether file servers, disk array controllers, or even IDE disks) run different software on separate hardware, as illustrated in Figure 1. This fact enables server-embedded security functionality that cannot be disabled by any software running on client systems (including the OS kernel). Further, storage devices often have fewer network interfaces (e.g., RPC+SNMP+HTTP or just SCSI) and no local users. Thus, compromising a storage server should be more difficult than compromising a client system. Of course, such servers have a limited view of system activity, so they cannot distinguish legitimate users from clever impostors. But, from behind the physical storage interface, a storage IDS can spot many common intruder actions and alert administrators.

Administrators must be able to communicate with the storage IDS, both to configure it and to receive alerts. This administrative channel must also be compromise-independent of client systems, meaning that no user (including root) and no software (including the OS kernel) on a client system can have administrative privileges for the storage IDS. Section 4 discusses deployment options for the administrative console, including physical consoles and cryptographic channels from a dedicated administrative system.

All of the warning signs discussed in this paper could also be spotted from within a HIDS, but host-based IDSs do not enjoy the compromise independence of storage IDSs. A host-based IDS is vulnerable to being disabled or bypassed by intruders that compromise the OS kernel. Another interesting place for a storage IDS is the virtual disk module of a virtual machine monitor [39]; such deployment would enjoy compromise independence from the OSes running in its virtual machines [4].

2.3 Warning signs for storage IDSs

Successful intruders often modify stored data. For instance, they may overwrite system utilities to hide their presence, install Trojan horse daemons to allow for reentry, add users, modify startup scripts to reinstall kernel modifications upon reboot, remove evidence from the audit log, or store illicit materials. These modifications are visible to the storage system when they are made persistent. This section describes four categories of warning signs that a storage IDS can monitor: data and attribute modifications, update patterns, content integrity, and suspicious content.

2.3.1 Data/attribute modification

In managed computing environments, the simplest (and perhaps most effective) category of warning signs consists of data or meta-data changes to files that administrators expect to remain unchanged except during explicit upgrades. Examples of such files include system executables and scripts, configuration files, and system header files and libraries. Given the importance of such files and the infrequency of updates to them, any modification is a potential sign of intrusion. A storage IDS can detect all such modifications on-the-fly, before the storage device processes each request, and issue an alert immediately.

In current systems, modification detection is sometimes provided by a checksumming utility (e.g., Tripwire [18]) that periodically compares the current storage state against a reference database stored elsewhere. Storage-based intrusion detection improves on this current approach in three ways: (1) it allows immediate detection of changes to watched files; (2) it can notice short-term changes, made and then undone, which would not be noticed by a checksumming utility if the changes occurred between two periodic checks; and (3) for local storage, it avoids trusting the host OS to perform the checks, which many rootkits disable or bypass.

2.3.2 Update patterns

A second category of warning signs consists of suspicious access patterns, particularly updates. There are several concrete examples for which storage IDSs can be useful in watching. The clearest is client system audit logs; these audit logs are critical to both intrusion detection [6] and diagnosis [35], leading many intruders to scrub evidence from them as a precaution. Any such manipulation will be obvious to a storage IDS that understands the well-defined update pattern of the specific audit log. For instance, audit log files are usually append-only, and they may be periodically "rotated." This rotation consists of renaming the current log file to an alternate name (e.g., `logfile` to `logfile.0`) and creating a new "current" log file. Any deviation in the update pattern of the current log file or any modification of a previous log file is suspicious.

Another suspicious update pattern is timestamp reversal. Specifically, the data modification and attribute change times commonly kept for each file can be quite useful for post-intrusion diagnosis of which files were manipulated [9]. By manipulating the times stored in inodes (e.g., setting them back to their original values), an intruder can inhibit such diagnosis. Of course, care must be taken with IDS rules, since some programs (e.g., `tar`) legitimately set these times to old values. One possibility would be to only set off an alert when the modification time is set back long after a file's creation. This would exclude `tar`-style activity but would catch an intruder trying to obfuscate a modified file. Of course, the intruder could now delete the file, create a new one, set the date back, and hide from the storage IDS—a more complex rule could catch this, but such escalation is the nature of intrusion detection.

Detection of storage denial-of-service (DoS) attacks also falls into the category of suspicious access patterns. For example, an attacker can disable specific services or entire systems by allocating all or most of the free space. A similar effect can be achieved by allocating inodes or other metadata structures. A storage IDS can watch for such exhaustion, which may be deliberate, accidental, or coincidental (e.g., a user just downloaded 10 GB of multimedia files). When the system reaches predetermined thresholds of unallocated resources and allocation rate, warning the administrator is appropriate even in non-intrusion situations—attention is likely to be necessary soon. A storage IDS could similarly warn the administrator when storage activity exceeds a threshold for too long, which may be a DoS attack or just an indication that the server needs to be upgraded.

Although specific rules can spot expected intruder actions, more general rules may allow larger classes of suspicious activity to be noticed. For example, some attribute modifications, like enabling "set UID" bits or reducing the permissions needed for access, may indicate foul play. Additionally, many applications access storage in a regular manner. As two examples: word processors often use temporary and backup files in specific ways, and UNIX password management involves a pair of inter-related files (`/etc/passwd` and `/etc/shadow`). The corresponding access patterns seen at the storage device will be a reflection of the application's requests. This presents an opportunity for anomaly detection based on how a given file is normally accessed. This could be done in a manner similar to learning common patterns of system calls [10] or starting with rules regarding the expected behavior of individual applications [19]. Deviation from the expected pattern could indicate an intruder attempting to subvert the normal method of accessing a given file. Of course, the downside is an increase (likely substantial) in the number of false alarms. Our focus to date has been on explicit detection rules, but anomaly detection within storage access patterns is an interesting topic for future research.

2.3.3 Content integrity

A third category of warning signs consists of changes that violate internal consistency rules of specific files. This category builds on the previous examples by understanding the application-specific semantics of particularly important stored data. Of course, to verify content integrity, the device must understand the format of a file. Further, while simple formats may be verified in the context of the write operation, file formats may be arbitrarily complex and verification may require access to additional data blocks (other than those currently being written). This creates a performance vs. security trade-off made by deciding which files to verify and how often to verify them. In practice, there are likely to be few critical files for which content integrity verification is utilized.

As a concrete example, consider a UNIX system password file (`/etc/passwd`), which consists of a set of well-defined records. Records are delimited by a line-break, and each record consists of seven colon-separated fields. Further, each of the fields has a specific meaning, some of which are expected to conform to rules of practice. For example, the seventh field specifies the "shell" program to be launched when a user logs in, and (in Linux) the file `/etc/shells` lists the legal options. During the "Capture the Flag" information warfare game at the 2002 DEF CON conference [21], one tactic used was to change the root shell on compromised systems to `/sbin/halt`; once a targeted system's administrator noted the intrusion and attempted to become root on the machine (the common initial reaction), considerable down-time and administrative effort was needed to restore the system to operation. A storage IDS can monitor changes to `/etc/passwd` and verify that they conform to a set of basic integrity rules: 7-field records, non-empty password field, legal default shell, legal home directory, non-overlapping user IDs, etc. The attack described above, among others, could be caught immediately.

2.3.4 Suspicious content

A fourth category of warning signs is the appearance of suspicious content. The most obvious suspicious content is a known virus or rootkit, detectable via its signature. Several high-end storage servers (e.g., from EMC [24] and Network Appliance [28]) now include support for internal virus scanning. By executing the scans within the storage server, viruses cannot disable the scanners even after infecting clients.

Two other examples of suspicious content are large numbers of "hidden" files or empty files. Hidden files have names that are not displayed by normal directory listing interfaces [7, p. 217], and their use may indicate that an intruder is using the system as a storage repository, perhaps for illicit or pirated content. A large number of empty files or directories may indicate an attempt to exploit a race condition [2, 30] by inducing a time-consuming directory listing, search, or removal.

2.4 Limitations, costs, and weaknesses

Although storage-based intrusion detection contributes to security efforts, of course it is not a silver bullet.

Like any IDS, a storage IDS will produce some false positives. With very specific rules, such as "watch these 100 files for any modification," false positives should be infrequent; they will occur only when there are legitimate

changes to a watched file, which should be easily verified if updates involve a careful procedure. The issue of false alarms grows progressively more problematic as the rules get less exact (e.g., the time reversal or resource exhaustion examples). The far end of the spectrum from specific rules is general anomaly detection.

Also like any IDS, a storage IDS will fail to spot some intrusions. Fundamentally, a storage IDS cannot notice intrusions whose actions do not cause odd storage behavior. For example, three of the eighteen intrusion tools examined in the next section modify the OS but change no files. Also, an intruder may manipulate storage in unwatched ways. Using network-based and host-based IDSs together with a storage IDS can increase the odds of spotting various forms of intrusion.

Intrusion detection, as an aspect of information warfare, is by nature a "game" of escalation. As soon as one side takes away an avenue of attack, the other starts looking for the next. Since storage-based intrusion detection easily sees several common intruder activities, crafty intruders will change tactics. For example, an intruder can make any number of changes to the host's memory, so long as those modifications do not propagate to storage. A reboot, however, will reset the system and remove the intrusion, which argues for proactive restart [3, 16, 43]. To counter this, attackers must have their changes re-established automatically after a reboot, such as by manipulating the various boot-time (e.g., `rc.local` in UNIX-like systems) or periodic (e.g., `cron` in UNIX-like systems) programs. Doing so exposes them to the storage IDS, creating a traditional intrusion detection game of cat and mouse.

As a practical consideration, storage IDSs embedded within individual components of decentralized storage systems are unlikely to be effective. For example, a disk array controller is a fine place for storage-based intrusion detection, but individual disks behind software striping are not. Each of the disks has only part of the file system's state, making it difficult to check non-trivial rules without adding new inter-device communication paths.

Finally, storage-based intrusion detection is not free. Checking rules comes with some cost in processing and memory resources, and more rules require more resources. In configuring a storage IDS, one must balance detection efforts with performance costs for the particular operating environment.

3 Case Studies

This section explores how well a storage IDS might fare in the face of actual compromises. To do so, we examined eighteen intrusion tools (Table 1) designed to be run on compromised systems. All were downloaded from public websites, most of them from Packet Storm [26].

Most of the actions taken by these tools fall into two categories. Actions in the first category involve hiding evidence of the intrusion and the rootkit's activity. The second provides a mechanism for reentry into a system. Twelve of the tools operate by running various binaries on the host system and overwriting existing binaries to continue gaining control. The other six insert code into the operating system kernel.

For the analysis in this section, we focus on a subset of the rules supported by our prototype storage-based IDS described in Section 5. Specifically, we include the file/directory modification (Tripwire-like) rules, the append-only logfile rule, and the hidden filename rules. We do not consider any "suspicious content" rules, which may or may not catch a rootkit depending on whether its particular signature is known.[1] In these eighteen toolkits, we did not find any instances of resource exhaustion attacks or of reverting inode times.

3.1 Detection results

Of the eighteen toolkits tested, storage IDS rules would immediately detect fifteen based on their storage modifications. Most would trigger numerous alerts, highlighting their presence. The other three make no changes to persistent storage. However, they are removed if the system reboots; all three modify the kernel, but would have to be combined with system file changes to be re-inserted upon reboot.

Non-append changes to the system audit log. Seven of the eighteen toolkits scrub evidence of system compromise from the audit log. All of them do so by selectively overwriting entries related to their intrusion into the system, rather than by truncating the logfile entirely. All cause alerts to be generated in our prototype.

System file modification. Fifteen of the eighteen toolkits modify a number of watched system files (ranging from 1 to 20). Each such modification generates an alert. Although three of the rootkits replace the files with binaries that match the size and CRC checksum of the previous files, they do not foil cryptographically-strong hashes. Thus, Tripwire-like systems would be able to catch them as well, though the evasion mechanism described in Section 3.2 defeats Tripwire.

Many of the files modified are common utilities for system administration, found in `/bin`, `/sbin`, and `/usr/bin` on a UNIX machine. They are modified to hide the presence and activity of the intruder. Common changes include

[1] An interesting note is that rootkit developers reuse code: four of the rootkits use the same audit log scrubbing program (`sauber`), and another three use a different program (`zap2`).

Name	Description	Syscall redir.	Log scrub	Hidden dirs	Watched files	Total alerts
Ramen	Linux worm			X	2	3
1ion	Linux worm				10	10
FK 0.4	Linux LKM rootkit and trojan ssh	X			1	1
Taskigt	Linux LKM rootkit				1	1
SK 1.3a	Linux kernel rootkit via `/dev/kmem`	X				-
Darkside 0.2.3	FreeBSD LKM rootkit	X				-
Knark 0.59	Linux LKM rootkit	X		X	1	2
Adore	Linux LKM rootkit	X				-
lrk5	User level rootkit from source		X	X	20	22
Sun rootkit	SunOS rootkit with trojan rlogin				1	1
FreeBSD Rootkit 2	User level FreeBSD rootkit		X	X	15	17
t0rn	Linux user level rootkit		X	X	20	22
Advanced Rootkit	Linux user level rootkit			X	10	11
ASMD	Rootkit w/SUID binary trojan			X	1	2
Dica	Linux user level rootkit		X	X	9	11
Flea	Linux user level rootkit		X	X	20	22
Ohara	Rootkit w/PAM trojan		X	X	4	6
TK 6.66	Linux user level rootkit		X	X	10	12

Table 1: **Visible actions of several intruder toolkits.** For each of the tools, the table shows which of the following actions are performed: redirecting system calls, scrubbing the system log files, and creating hidden directories. It also shows how many of the files watched by our rule set are modified by a given tool. The final column shows the total number of alerts generated by a given tool.

modifying ps to not show an intruder's processes, `ls` to not show an intruder's files, and `netstat` to not show an intruder's open network ports and connections. Similar modifications are often made to `grep`, `find`, `du`, and `pstree`.

The other common reason for modifying system binaries is to create backdoors for system reentry. Most commonly, the target is `telnetd` or `sshd`, although one rootkit added a backdoored PAM module [33] as well. Methods for using the backdoor vary and do not impact our analysis.

Hidden file or directory names. Twelve of the rootkits make a hard-coded effort to hide their non-executable and working files (i.e., the files that are not replacing existing files). Ten of the kits use directories starting in a '.' to hide from default `ls` listings. Three of these generate alerts by trying to make a hidden directory look like the reserved '.' or '..' directories by appending one or more spaces ('. ' or '.. '). This also makes the path harder to type if a system administrator does not know the number of spaces.

3.2 Kernel-inserted evasion techniques

Six of the eighteen toolkits modified the running operating system kernel. Five of these six "kernel rootkits" include loadable kernel modules (LKMs), and the other inserts itself directly into kernel memory by use of the `/dev/kmem` interface. Most of the kernel modifications allow intruders to hide as well as reenter the system, similarly to the file modifications described above. Especially interesting for this analysis is the use of `exec()` redirection by four of the kernel rootkits. With such redirection, the `exec()` system call uses a replacement version of a targeted program, while other system calls return information about or data from the original. As a result, any tool relying on the accuracy of system calls to check file integrity, such as Tripwire, will be fooled.

All of these rootkits are detected using our storage IDS rules—they all put their replacement programs in the originals' directories (which are watched), and four of the six actually move the original file to a new name and store their replacement file with the original name (which also triggers an alert). However, future rootkits could be modified to be less obvious to a storage IDS. Specifically, the original files could be left untouched and replacement files could be stored someplace not watched by the storage IDS, such as a random user directory—neither would generate an alert. With this approach, file modification can be completely hidden from a storage IDS unless the rootkit wants to reinstall the kernel modification after a reboot. To accomplish this, some original files would need to be changed, which forces intruders to make an interesting choice: hide from the storage IDS or persist beyond the next reboot.

3.3 Anecdotal experience

During the writing of this paper, one of the authors happened to be asked to analyze a system that had been recently compromised. Several modifications similar to those made by the above rootkits were found on the system. Root's `.bash_profile` was modified to run the zap2 log scrubber, so that as soon as root logged into the system to investigate the intrusion, the related logs would be scrubbed. Several binaries were modified (`ps`, `top`, `netstat`, `pstree`, `sshd`, and `telnetd`). The binaries were setup to hide the existence of an IRC bot, running out of the directory `'/dev/.. /'`. This experience helps validate our choice of "rootkits" for study, as they appear to be representative of at least one real-world intrusion. This intrusion would have triggered at least 8 storage IDS rules.

4 Design of a Storage IDS

To be useful in practice, a storage IDS must simultaneously achieve several goals. It must support a useful set of detection rules, while also being easy for human administrators to understand and configure. It must be efficient, minimizing both added delay and added resource requirements; some user communities still accept security measures only when they are "free." Additionally, it should be invisible to users at least until an intrusion detection rule is matched.

This section describes four aspects of storage IDS design: specifying detection rules, administering a storage IDS securely, checking detection rules, and responding to suspicious activity.

4.1 Specifying detection rules

Specifying rules for an IDS is a tedious, error prone activity. The tools an administrator uses to write and manipulate those rules should be as simple and straightforward as possible. Each of the four categories of suspicious activity presented earlier will likely need a unique format for rule specification.

The rule format used by Tripwire seems to work well for specifying rules concerned with data and attribute modification. This format allows an administrator to specify the pathname of a file and a list of properties that should be monitored for that file. The set of watchable properties are codified, and they include most file attributes. This rule language works well, because it allows the administrator to manipulate a well understood representation (pathnames and files), and the list of attributes that can be watched is small and well-defined.

The methods used by virus scanners work well for configuring an IDS to look for suspicious content. Rules can be specified as signatures that are compared against files' contents. Similarly, filename expression grammars (like those provided in scripting languages) could be used to describe suspicious filenames.

Less guidance exists for the other two categories of warning signs: update patterns and content integrity. We do not currently know how to specify general rules for these categories. Our approach has been to fall back on Tripwire-style rules; we hard-code checking functions (e.g., for non-append update or a particular content integrity violation) and then allow an administrator to specify on which files they should be checked (or that they should be checked for every file). More general approaches to specifying detection rules for these categories of warning signs are left for future work.

4.2 Secure administration

The security administrator must have a secure interface to the storage IDS. This interface is needed for the administrator to configure detection rules and to receive alerts. The interface must prevent client systems from forging or blocking administrative requests, since this could allow a crafty intruder to sneak around the IDS by disarming it. At a minimum, it must be tamper-evident. Otherwise, intruders could stop rule updates or prevent alerts from reaching the administrator. To maintain compromise independence, it must be the case that obtaining "superuser" or even kernel privileges on a client system is insufficient to gain administrative access to the storage device.

Two promising architectures exist for such administration: one based on physical access and one based on cryptography. For environments where the administrator has physical access to the device, a local administration terminal that allows the administrator to set detection rules and receive the corresponding alert messages satisfies the above goals.

In environments where physical access to the device is not practical, cryptography can be used to secure communications. In this scenario, the storage device acts as an endpoint for a cryptographic channel to the administrative system. The device must maintain keys and perform the necessary cryptographic functions to detect modified messages, lost messages, and blocked channels. Architectures for such trust models in storage systems exist [14]. This type of infrastructure is already common for administration of other network-attached security components, such as firewalls or network intrusion detection systems. For direct-attached storage devices, cryptographic channels can be used to tunnel administrative requests and alerts through the OS of the host system, as illustrated in Figure 2. Such tunneling simply treats the host OS as an

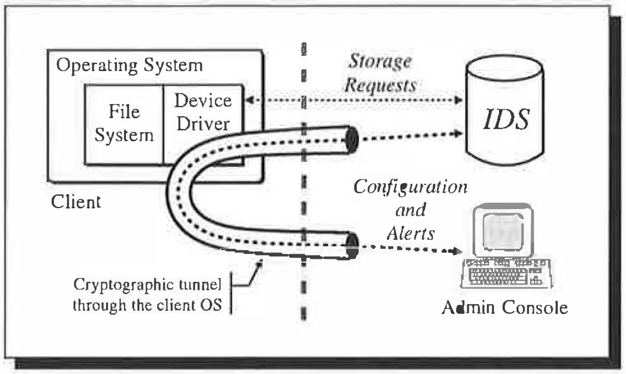


Figure 2: **Tunneling administrative commands through client systems.** For storage devices attached directly to client systems, a cryptographic tunnel can allow the administrator to securely manage a storage IDS. This tunnel uses the untrusted client OS to transport administrative commands and alerts.

untrusted network component.

For small numbers of dedicated servers in a machine room, either approach is feasible. For large numbers of storage devices or components operating in physically insecure environments, cryptography is the only viable solution.

4.3 Checking the detection rules

Checking detection rules can be non-trivial, because rules generally apply to full pathnames rather than inodes. Additional complications arise because rules can watch for files that do not yet exist.

For simple operations that act on individual files (e.g., READ and WRITE), rule verification is localized. The device need only check that the rules pertaining to that specific file are not violated (usually a simple flag comparison, sometimes a content check). For operations that affect the file system's namespace, verification is more complicated. For example, a rename of a directory tree may impact a large number of individual files, any of which could have IDS rules that must be checked. Renaming a directory requires examining all files and directories that are children of the one being renamed.

In the case of rules pertaining to files that do not currently exist, this list of rules must be consulted when operations change the namespace. For example, the administrator may want to watch for the existence of a file named `/a/b/c` even if the directory named `/a` does not yet exist. However, a single file system operation (e.g., `mv /z /a`) could cause the watched file to suddenly exist, given the appropriate structure for z's directory tree.

4.4 Responding to rule violations

Since a detected "intruder action" may actually be legitimate user activity (i.e., a false alarm), our default response is simply to send an alert to the administrative system or the designated alert log file. The alert message should contain such information as the file(s) involved, the time of the event, the action being performed, the action's attributes (e.g., the data written into the file), and the client's identity. Note that, if the rules are set properly, most false positives should be caused by legitimate updates (e.g., upgrades) from an administrator. With the right information in alerts, an administrative system that also coordinates legitimate upgrades could correlate the generated alert (which can include the new content) with the in-progress upgrade; if this were done, it could prevent the false alarm from reaching the human administrator while simultaneously verifying that the upgrade went through to persistent storage correctly.

There are more active responses that a storage IDS could trigger upon detecting suspicious activity. When choosing a response policy, of course, the administrator must weigh the benefits of an active response against the inconvenience and potential damage caused by false alarms.

One reasonable active response is to slow down the suspected intruder's storage accesses. For example, a storage device could wait until the alert is acknowledged before completing the suspicious request. It could also artificially increase request latencies for a client or user that is suspected of foul play. Doing so would provide increased time for a more thorough response, and, while it will cause some annoyance in false alarm situations, it is unlikely to cause damage. The device could even deny a request entirely if it violates one of the rules, although this response to a false alarm could cause damage and/or application failure. For some rules, like append-only audit logs, such access control may be desirable.

Liu, et al. proposed a more radical response to detected intrusions: isolating intruders, via versioning, at the file system level [22]. To do so, the file system forks the version trees to sandbox suspicious users until the administrator verifies the legitimacy of their actions. Unfortunately, such forking is likely to interfere with system operation, unless the intrusion detection mechanism yields no false alarms. Specifically, since suspected users modify different versions of files from regular users, the system faces a difficult reintegration [20, 41] problem, should the updates be judged legitimate. Still, it is interesting to consider embedding this approach, together with a storage IDS, into storage systems for particularly sensitive environments.

A less intrusive storage-embedded response is to start versioning all data and auditing all storage requests when an intrusion is detected. Doing so provides the administra-

tor with significant information for post-intrusion diagnosis and recovery. Of course, some intrusion-related information will likely be lost unless the intrusion is detected immediately, which is why Strunk et al. [38] argue for always doing these things (just in case). Still, IDS-triggered employment of this functionality may be a useful trade-off point.

5 Storage-based intrusion detection in an NFS server

To explore the concepts and feasibility of storage-based intrusion detection, we implemented a storage IDS in an NFS server. Unmodified client systems access the server using the standard NFS version 2 protocol [40][2], while storage-based intrusion detection occurs transparently. This section describes how the prototype storage IDS handles detection rule specification, the structures and algorithms for checking rules, and alert generation.

The base NFS server is called S4, and its implementation is described and evaluated elsewhere [38]. It internally performs file versioning and request auditing, using a log-structured file system [32], but these features are not relevant here. For our purposes, it is a convenient NFS file server with performance comparable to the Linux and FreeBSD NFS servers. Secure administration is performed via the server's console, using the physical access control approach.

5.1 Specifying detection rules

Our prototype storage IDS is capable of watching for a variety of data and metadata changes to files. The administrator specifies a list of Tripwire-style rules to configure the detection system. Each administrator-supplied rule is of the form: {*pathname, attribute-list*}—designating which attributes to monitor for a particular file. The list of attributes that can be watched is shown in Table 2. In addition to standard Tripwire rules, we have added two additional functions that can be specified on a per-file basis. The first watches for non-append changes, as described earlier; any truncation or write anywhere but at the previous end of a file will generate an alert. The second checks a file's integrity against the password file integrity rule discussed earlier. After every write, the file must conform to the rigid structure of a password file (7 colons per line), and all of the shells must be contained in the "acceptable" list.

In addition to per-file rules, an administrator can choose to

[2]The use of the NFSv2 protocol is an artifact of the server implementation the IDS is built into, but makes no difference in the areas we care about.

Metadata	
• inode modification time	• data modification time
• access time	• file permissions
• link count	• device number
• file owner	• inode number
• file type	• file owner group
• file size	
Data	
• any modification	• append only
• password structure	

Table 2: **Attribute list.** Rules can be established to watch these attributes in real-time on a file-by-file basis.

enable any of three system-wide rules: one that matches on any operation that rolls-back a file's modification time, one that matches on any operation that creates a "hidden" directory (e.g., a directory name beginning with '.' and having spaces in it), and one that looks for known (currently hard-coded) intrusion tools by their sizes and SHA-1 digests. Although the system currently checks the digests on every file update, periodic scanning of the system would likely be more practical. These rules apply to all parts of the directory hierarchy and are specified as simply ON or OFF.

Rules are communicated to the server through the use of an administrative RPC. This RPC interface has two commands (see Table 3). The `setRule()` RPC gives the IDS two values: the path of the file to be watched, and a set of flags describing the specific rules for that file. Rules are removed through the same mechanism, specifying the path and an empty rule set.

5.2 Checking the detection rules

This subsection describes the core of the storage IDS. It discusses how rules are stored and subsequently checked during operation.

5.2.1 Data structures

Three new structures allow the storage IDS to efficiently support the detection rules: the reverse lookup table, the inode watch flags, and the non-existent names table.

Reverse lookup table: The reverse lookup table serves two functions. First, it serves as a list of rules that the server is currently enforcing. Second, it maps an inode number to a pathname. The alert generation mechanism uses the latter to provide the administrator with file names instead of inode numbers, without resorting to a brute-force search of the namespace.

The reverse lookup table is populated via the `setRule()`

Command	Purpose	Direction
`setRule(path, rules)`	Changes the watched characteristics of a file. This command is used to both set and delete rules.	admin⇒server
`listRules()`	Retrieves the server's rule table as a list of {`pathname`, `rules`} records.	admin⇒server
`alert(path, rules, operation)`	Delivers a warning of a rule violation to the administrator.	server⇒admin

Table 3: **Administrative commands for our storage IDS.** This table lists the small set of administrative commands needed for an administrative console to configure and manage the storage IDS. The first two are sent by the console, and the third is sent by the storage IDS. The pathname refers to a file relative to the root of an exported file system. The *rules* are a description of what to check for, which can be any of the changes described in Table 2. The *operation* is the NFS operation that caused the rule violation.

RPC. Each rule's full pathname is broken into its component names, which are stored in distinct rows of the table. For each component, the table records four fields: *inode-number*, *directory-inode-number*, *name*, and *rules*. Indexed by *inode-number*, an entry contains the *name* within a parent directory (identified by its *directory-inode-number*). The *rules* associated with this *name* are a bitmask of the attributes and patterns to watch. Since a particular inode number can have more than one name, multiple entries for each inode may exist. A given inode number can be translated to a full pathname by looking up its lowest-level name and recursively looking up the name of the corresponding directory inode number. The search ends with the known inode number of the root directory. All names for an inode can be found by following all paths given by the lookup of the inode number.

Inode `watchflags` field: During the `setRule()` RPC, in addition to populating the reverse lookup table, a rule mask of 16 bits is computed and stored in the `watchflags` field of the watched file's inode. Since multiple pathnames may refer to the same inode, there may be more than one rule for a given file, and the mask contains the union. The inode `watchflags` field is a performance enhancement designed to co-locate the rules governing a file with that file's metadata. This field is not necessary for correctness since the pertinent data could be read from the reverse lookup table. However, it allows efficient verification of detection rules during the processing of an NFS request. Since the inode is read as part of any file access, most rule checking becomes a simple mask comparison.

Non-existent names table: The non-existent names table lists rules for pathnames that do not currently exist. Each entry in the table is associated with the deepest-level (existing) directory within the pathname of the original rule. Each entry contains three fields: *directory-inode-number*, *remaining-path*, and *rules*. Indexed by *directory-inode-number*, an entry specifies the *remaining-path*. When a file or directory is created or removed, the non-existent names table is consulted and updated, if necessary. For example, upon creation of a file for which a detection rule exists, the *rules* are checked and inserted in the `watchflags` field of the inode. Together, the reverse lookup table and the non-existent names table contain the entire set of IDS rules in effect.

5.2.2 Checking rules during NFS operations

We now describe the flow of rule checking, much of which is diagrammed in Figure 3, in two parts: changes to individual files and changes to the namespace.

Checking rules on individual file operations: For each NFS operation that affects only a single file, a mask of rules that might be violated is computed. This mask is compared, bitwise, to the corresponding `watchflags` field in the file's inode. For most of the rules, this comparison quickly determines if any alerts should be triggered. If the "password file" or "append only" flags are set, the corresponding verification function executes to determine if the rule is violated.

Checking rules on namespace operations: Namespace operations can cause watched pathnames to appear or disappear, which will usually trigger an alert. For operations that create watched pathnames, the storage IDS moves rules from the non-existent names table to the reverse lookup table. Conversely, operations that delete watched pathnames cause rules to move between tables in the opposite direction.

When a name is created (via CREATE, MKDIR, LINK, or SYMLINK) the non-existent names table is checked. If there are rules for the new file, they are checked and placed in the `watchflags` field of the new inode. In addition, the corresponding rule is removed from the non-existent names table and is added to the reverse lookup table. During a MKDIR, any entries in the non-existent names table that include the new directory as the next step in their remaining path are replaced; the new entries are indexed by the new directory's inode number and its name is removed from the remaining path.

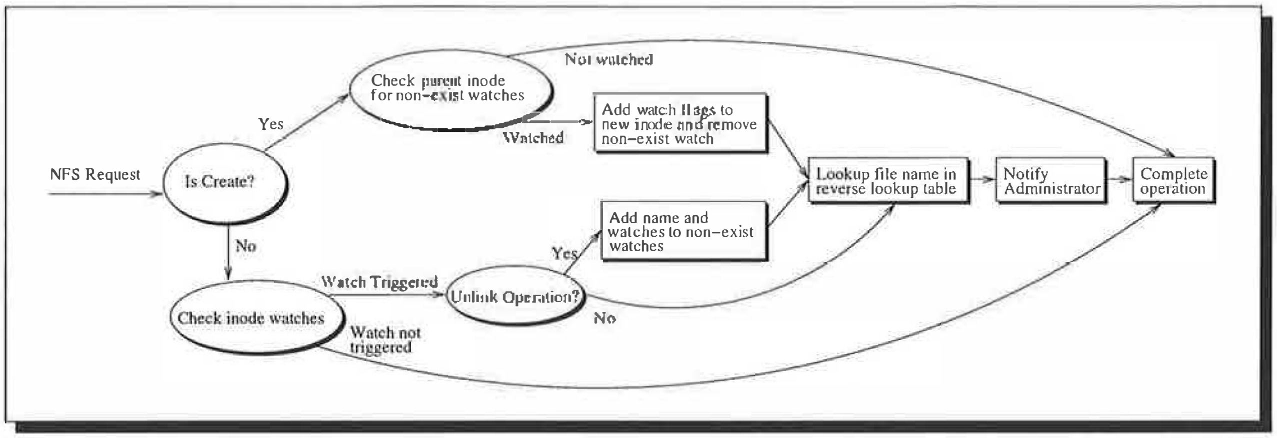


Figure 3: **Flowchart of our storage IDS.** Few structures and decision points are needed. In the common case (no rules for the file), only one inode's `watchflags` field is checked. The picture does not show RENAME operations here due to their complexity.

When a name is removed (via UNLINK or RMDIR), the `watchflags` field of the corresponding inode is checked for rules. Most such rules will trigger an alert, and an entry for them is also added to the non-existent names table. For RMDIR, the reverses of the actions for MKDIR are necessary. Any non-existent table entries parented on the removed directory must be modified. The removed directory's name is added to the beginning of each remaining path, and the directory inode number in the table is modified to be the directory's parent.

By far, the most complex namespace operation is a RENAME. For a RENAME of an individual file, modifying the rules is the same as a CREATE of the new name and a REMOVE of the old. When a directory is renamed, its subtrees must be recursively checked for watched files. If any are found, and once appropriate alerts are generated, their rules and pathname up to the parent of the renamed directory are stored in the non-existent names table, and the `watchflags` field of the inode is cleared. Then, the non-existent names table must be checked (again recursively) for any rules that map into the directory's new name and its children; such rules are checked, added to the inode's `watchflags` field, and updated as for name creation.

5.3 Generating alerts

Alerts are generated and sent immediately when a detection rule is triggered. The alert consists of the original detection rule (pathname and attributes watched), the specific attributes that were affected, and the RPC operation that triggered the rule. To get the original rule information, the reverse lookup table is consulted. If a single RPC operation triggers multiple rules, one alert is sent for each.

5.4 Storage IDS rules in a NIDS

Because NFS traffic goes over a traditional network, the detection rules described for our prototype storage IDS could be implemented in a NIDS. However, this would involve several new costs. First, it would require the NIDS to watch the LAN links that carry NFS activity. These links are usually higher bandwidth than the Internet uplinks on which most NIDSs are used.[3] Second, it would require that the NIDS replicate a substantial amount of work already performed by the NFS server, increasing the CPU requirements relative to an in-server storage IDS. Third, the NIDS would have to replicate and hold substantial amounts of state (e.g. mappings of file handles to their corresponding files). Our experiences checking rules against NFS traces indicate that this state grows rapidly because the NFS protocol does not expose to the network (or the server) when such state can be removed. Even simple attribute updates cannot be checked without caching the old values of the attributes, otherwise the NIDS could not distinguish modified attributes from reapplied values. Fourth, rules cannot always be checked by looking only at the current command. The NIDS may need to read file data and attributes to deal with namespace operations, content integrity checks, and update pattern rules. In addition to the performance penalty, this requires giving the NIDS read permission for all NFS files and directories.

Given all of these issues, we believe that embedding storage IDS checks directly into the storage component is more appropriate.

[3]Tapping a NIDS into direct-attached storage interconnects, such as SCSI and FibreChannel, would be more difficult.

6 Evaluation

This section evaluates the costs of our storage IDS in terms of performance impact and memory required—both costs are minimal.

6.1 Experimental setup

All experiments use the S4 NFS server, with and without the new support for storage-based intrusion detection. The client system is a dual 1 GHz Pentium III with 128 MB RAM and a 3Com 3C905B 100 Mbps network adapter. The server is a dual 700 MHz Pentium III with 512 MB RAM, a 9 GB 10,000 RPM Quantum Atlas 10K II drive, an Adaptec AIC-7896/7 Ultra2 SCSI controller, and an Intel EtherExpress Pro 100 Mb network adapter. The client and server are on the same 100 Mb network switch. The operating system on all machines is Red Hat Linux 6.2 with Linux kernel version 2.2.14.

SSH-build was constructed as a replacement for the Andrew file system benchmark [15, 36]. It consists of 3 phases: The unpack phase, which unpacks the compressed tar archive of SSH v. 1.2.27 (approximately 1 MB in size before decompression), stresses metadata operations on files of varying sizes. The configure phase consists of the automatic generation of header files and makefiles, which involves building various small programs that check the existing system configuration. The build phase compiles, links, and removes temporary files. This last phase is the most CPU intensive, but it also generates a large number of object files and a few executables. Both the server and client caches are flushed between phases.

PostMark was designed to measure the performance of a file system used for electronic mail, netnews, and web based services [17]. It creates a large number of small randomly-sized files (between 512 B and 16 KB) and performs a specified number of transactions on them. Each transaction consists of two sub-transactions, with one being a create or delete and the other being a read or append. The default configuration used for the experiments consists of 100,000 transactions on 20,000 files, and the biases for transaction types are equal.

6.2 Performance impact

The storage IDS checks a file's rules before any operation that could possibly trigger an alert. This includes READ operations, since they may change a file's last access time. Additionally, namespace-modifying operations require further checks and possible updates of the nonexistent names table. To understand the performance consequences of the storage IDS design, we ran PostMark and SSH-Build tests. Since our main concern is avoiding a performance loss in the case where no rule is violated, we ran these benchmarks with no relevant rules set. As long as no rules match, the results are similar with 0 rules, 1000 rules on existing files, or 1000 rules on non-existing files. Table 4 shows that the performance impact of the storage IDS is minimal. The largest performance difference is for the configure and build phases of SSH-build, which involve large numbers of namespace operations.

Benchmark	Baseline	With IDS	Change
SSH untar	27.3 (0.02)	27.4 (0.02)	0.03%
SSH config.	42.6 (0.68)	43.2 (0.37)	1.3%
SSH build	85.9 (0.18)	86.8 (0.17)	1.0%
PostMark	4288 (11.9)	4290 (13.0)	0.04%

Table 4: **Performance of macro benchmarks.** All benchmarks were run with and without the storage IDS functionality. Each number represents the average of 10 trials in seconds (with the standard deviation in parenthesis).

Benchmark	Baseline	With IDS	Change
Create	4.32	4.35	0.7%
Remove	4.50	4.65	3.3%
Mkdir	4.36	4.38	0.5%
Rmdir	4.52	4.59	1.5%
Rename file	3.81	3.91	2.6%
Rename dir	3.91	4.04	3.3%

Table 5: **Performance of micro benchmarks.** All benchmarks were run with and without the storage IDS functionality. Each number represents the average of 1000 trials in milliseconds.

Microbenchmarks on specific filesystem actions help explain the overheads. Table 5 shows results for the most expensive operations, which all affect the namespace. The performance differences are caused by redundancy in the implementation. The storage IDS code is kept separate from the NFS server internals, valuing modularity over performance. For example, name removal operations involve a redundant directory lookup and inode fetch (from cache) to locate the corresponding inode's `watchflags` field.

Rules take very little time to generate alerts. For example, a write to a file with a rule set takes 4.901 milliseconds if no alert is set off. If an alert is set off the time is 4.941 milliseconds. These represent the average over 1000 trials, and show a .8% overhead.

6.3 Space efficiency

The storage IDS structures are stored on disk. To avoid extra disk accesses for most rule checking, though, it is important that they fit in memory.

Three structures are used to check a set of rules. First, each inode in the system has an additional two-byte field for the bitmask of the rules on that file. There is no cost for this, because the space in the inode was previously unused. Linux's ext2fs and BSD's FFS also have sufficient unused space to store such data without increasing their inode sizes. If space were not available, the reverse lookup table can be used instead, since it provides the same information. Second, for each pathname component of a rule, the reverse lookup table requires $20 + N$ bytes: a 16-byte inode number, 2 bytes for the rule bitmask, and $N+2$ bytes for a pathname component of length N. Third, the non-existent names table contains one entry for every file being watched that does not currently exist. Each entry consumes 274 bytes: a 16-byte inode number, 2 bytes for the rule bitmask, and 256 bytes for the maximum pathname supported.

To examine a concrete example of how an administrator might use this system, we downloaded the open source version of Tripwire [42]. Included with it is an example rule file for Linux, containing (after expanding directories to lists of files) 4730 rules. We examined a Red Hat Linux 6.1 [31] desktop machine to obtain an idea of the number of watched files that actually exist on the hard drive. Of the 4730 watched files, 4689 existed on our example system. Using data structure sizes from above, reverse lookup entries for the watched files consume 141 KB. Entries in the non-existent name table for the remaining 41 watched files consume 11 KB. In total, only 152 KB are needed for the storage IDS.

6.4 False positives

We have explored the false positive rate of storage-based intrusion detection in several ways.

To evaluate the file watch rules, two months of traces of all file system operations were gathered on a desktop machine in our group. We compared the files modified on this system with the watched file list from the open source version of Tripwire. This uncovered two distinct patterns where files were modified. Nightly, the user list (`/etc/passwd`) on the machine was overwritten by a central server. Most nights it does not change but the create and rename performed would have triggered an alert. Additionally, multiple binaries in the system were replaced over time by the administrative upgrade process. In only one case was a configuration file on the system changed by a local user.

For alert-triggering modifications arising from explicit administrative action, a storage IDS can provide an added benefit. If an administrator pre-informs the admin console of updated files before they are distributed to machines, the IDS can verify that desired updates happen correctly. Specifically, the admin console can read the new contents via the admin channel and verify that they are as intended. If so, the update is known to have succeeded, and the alert can be suppressed.

We have also performed two (much) smaller studies. First, we have evaluated our "hidden filename" rules by examining the entire filesystems of several desktops and servers—we found no uses of any of them, including the '.' or '..' followed by any number of spaces discussed above. Second, we evaluated our "inode time reversal" rules by examining lengthy traces of NFS activity from our environment and from two Harvard environments [8]—we found a sizable number of false positives, caused mainly by unpacking archives with utilities like `tar`. Combined with the lack of time reversal in any of the toolkits, use of this rule may be a bad idea.

7 Additional Related Work

Much related work has been discussed within the flow of the paper. For emphasis, we note that there have been many intrusion detection systems focused on host OS activity and network communication; Axelsson [1] recently surveyed the state-of-the-art. Also, the most closely related tool, Tripwire [18], was used as an initial template for our prototype's file modification detection ruleset.

Our work is part of a recent line of research exploiting physical [12, 44] and virtual [4] protection boundaries to detect intrusions into system software. Notably, Garfinkel et al. [13] explore the utility of an IDS embedded in a virtual machine monitor (VMM), which can inspect machine state while being compromise independent of most host software. Storage-based intrusion detection rules could be embedded in a VMM's storage module, rather than in a physical storage device, to identify suspicious storage activity.

Perhaps the most closely related work is the original proposal for self-securing storage [38], which argued for storage-embedded support for intrusion survival. Self-securing storage retains every version of all data and a log of all requests for a period of time called the *detection window*. For intrusions detected within this window, security administrators have a wealth of information for post-intrusion diagnosis and recovery.

Such versioning and auditing complements storage-based intrusion detection in several additional ways. First, when creating rules about storage activity for use in detection, administrators can use the latest audit log and version history to test new rules for false alarms. Second, the audit log could simplify implementation of rules looking for patterns of requests. Third, administrators can use the history to investigate alerts of suspicious behavior (i.e., to check for supporting evidence within the history). Fourth, since

the history is retained, a storage IDS can delay checks until the device is idle, allowing the device to avoid performance penalties for expensive checks by accepting a potentially longer detection latency.

8 Conclusions and Future Work

A storage IDS watches system activity from a new viewpoint, which immediately exposes some common intruder actions. Running on separate hardware, this functionality remains in place even when client OSes or user accounts are compromised. Our prototype storage IDS demonstrates both feasibility and efficiency within a file server. Analysis of real intrusion tools indicates that most would be immediately detected by a storage IDS. After adjusting for storage IDS presence, intrusion tools will have to choose between exposing themselves to detection or being removed whenever the system reboots.

In continuing work, we are developing a prototype storage IDS embedded in a device exporting a block-based interface (SCSI). To implement the same rules as our augmented NFS server, such a device must be able to parse and traverse the on-disk metadata structures of the file system it holds. For example, knowing whether `/usr/sbin/sshd` has changed on disk requires knowing not only whether the corresponding data blocks have changed, but also whether the inode still points to the same blocks and whether the name still translates to the same inode. We have developed this translation functionality for two popular file systems, Linux's ext2fs and FreeBSD's FFS. The additional complexity required is small (under 200 lines of C code for each), simple (under 3 days of programming effort each), and changes infrequently (about 5 years between incompatible changes to on-disk structures). The latter, in particular, indicates that device vendors can deploy firmware and expect useful lifetimes that match the hardware. Sivathanu et al. [37] have evaluated the costs and benefits of device-embedded FS knowledge more generally, finding that it is feasible and valuable.

Another continuing direction is exploration of less exact rules and their impact on detection and false positive rates. In particular, the potential of pattern matching rules and general anomaly detection for storage remains unknown.

Acknowledgments

We thank the members and companies of the PDL Consortium (including EMC, Hewlett-Packard, Hitachi, IBM, Intel, Microsoft, Network Appliance, Oracle, Panasas, Seagate, Sun, and Veritas) for their interest, insights, feedback, and support. We thank IBM and Intel for hardware grants supporting our research efforts. This material is based on research sponsored by the Air Force Research Laboratory, under agreement number F49620-01-1-0433, and by DARPA/ITO's OASIS program, under Air Force contract number F30602-99-2-0539-AFRL.[4] Craig Soules was supported by a USENIX Fellowship. Garth Goodson was supported by an IBM Fellowship.

References

[1] S. Axelsson. *Research in intrusion-detection systems: a survey*. Technical report 98–17. Department of Computer Engineering, Chalmers University of Technology, December 1998.

[2] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing Systems*, **9**(2):131–152, Spring 1996.

[3] M. Castro and B. Liskov. Proactive recovery in a Byzantine-fault-tolerant system. Symposium on Operating Systems Design and Implementation, pages 273–287. USENIX Association, 2000.

[4] P. M. Chen and B. D. Noble. When virtual is better than real. Hot Topics in Operating Systems, pages 133–138. IEEE Comput. Soc., 2001.

[5] B. Cheswick and S. Bellovin. *Firewalls and Internet security: repelling the wily hacker*. Addison-Wesley, Reading, Mass. and London, 1994.

[6] D. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, **SE-13**(2):222–232, February 1987.

[7] D. E. Denning. *Information warfare and security*. Addison-Wesley, 1999.

[8] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS tracing of an email and research workload. Conference on File and Storage Technologies, pages 203–217. USENIX Association, 2003.

[9] D. Farmer. What are MACtimes? *Dr. Dobb's Journal*, **25**(10):68–74, October 2000.

[10] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for UNIX processes. IEEE Symposium on Security and Privacy, pages 120–128. IEEE, 1996.

[11] G. R. Ganger, G. Economou, and S. M. Bielski. *Finding and Containing Enemies Within the Walls with Self-securing Network Interfaces*. Carnegie Mellon University Technical Report CMU-CS-03-109. January 2003.

[12] G. R. Ganger and D. F. Nagle. Better security via smarter devices. Hot Topics in Operating Systems, pages 100–105. IEEE, 2001.

[4]The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Laboratory or the U.S. Government.

[13] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. NDSS. The Internet Society, 2003.

[14] H. Gobioff. *Security for a high performance commodity storage subsystem.* PhD thesis, published as TR CMU–CS–99–160. Carnegie-Mellon University, Pittsburgh, PA, July 1999.

[15] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, **6**(1):51–81, February 1988.

[16] Y. N. Huang, C. M. R. Kintala, L. Bernstein, and Y. M. Wang. Components for software fault-tolerance and rejuvenation. *AT&T Bell Laboratories Technical Journal*, **75**(2):29–37, March-April 1996.

[17] J. Katcher. *PostMark: a new file system benchmark.* Technical report TR3022. Network Appliance, October 1997.

[18] G. H. Kim and E. H. Spafford. The design and implementation of Tripwire: a file system integrity checker. Conference on Computer and Communications Security, pages 18–29. ACM, 1994.

[19] C. Ko, M. Ruschitzka, and K. Levitt. Execution monitoring of security-critical programs in distributed systems: a specification-based approach. IEEE Symposium on Security and Privacy, pages 175–187. IEEE, 1997.

[20] P. Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. USENIX Annual Technical Conference, pages 95–106. USENIX Association, 1995.

[21] R. Lemos. Putting fun back into hacking. ZDNet News, 5 August 2002. `http://zdnet.com.com/2100-1105-948404.html`.

[22] P. Liu, S. Jajodia, and C. D. McCollum. Intrusion confinement by isolation in information systems. IFIP Working Conference on Database Security, pages 3–18. IFIP, 2000.

[23] T. F. Lunt and R. Jagannathan. A prototype real-time intrusion-detection expert system. IEEE Symposium on Security and Privacy, pages 59–66. IEEE, 1988.

[24] McAfee NetShield for Celerra. EMC Corporation, August 2002. `http://www.emc.com/pdf/partnersalliances/einfo/McAfee_netshield.pdf`.

[25] NFR Security. `http://www.nfr.net/`, August 2002.

[26] Packet Storm Security. Packet Storm, 26 January 2003. `http://www.packetstormsecurity.org/`.

[27] V. Paxson. Bro: a system for detecting network intruders in real-time. USENIX Security Symposium, pages 31–51. USENIX Association, 1998.

[28] J. Phillips. *Antivirus scanning best practices guide.* Technical report 3107. Network Appliance Inc. `http://www.netapp.com/tech_library/3107.html`.

[29] P. A. Porras and P. G. Neumann. EMERALD: event monitoring enabling responses to anomalous live disturbances. National Information Systems Security Conference, pages 353–365, 1997.

[30] W. Purczynski. GNU fileutils – recursive directory removal race condition. BugTraq mailing list, 11 March 2002.

[31] Red Hat Linux 6.1, 4 March 1999. `ftp://ftp.redhat.com/pub/redhat/linux/6.1/`.

[32] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, **10**(1):26–52. ACM Press, February 1992.

[33] V. Samar and R. J. Schemers III. *Unified login with pluggable authentication modules (PAM).* Open Software Foundation RFC 86.0. Open Software Foundation, October 1995.

[34] J. Scambray, S. McClure, and G. Kurtz. *Hacking exposed: network security secrets & solutions.* Osborne/McGraw-Hill, 2001.

[35] B. Schneier and J. Kelsey. Secure audit logs to support computer forensics. *ACM Transactions on Information and System Security*, **2**(2):159–176. ACM, May 1999.

[36] M. I. Seltzer, G. R. Ganger, M. K. McKusick, K. A. Smith, C. A. N. Soules, and C. A. Stein. Journaling versus Soft Updates: Asynchronous Meta-data Protection in File Systems. USENIX Annual Technical Conference, 2000.

[37] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-smart disk systems. Conference on File and Storage Technologies, pages 73–89. USENIX Association, 2003.

[38] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-securing storage: protecting data in compromised systems. Symposium on Operating Systems Design and Implementation, pages 165–180. USENIX Association, 2000.

[39] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. USENIX Annual Technical Conference, pages 1–14. USENIX Association, 2001.

[40] Sun Microsystems. *NFS: network file system protocol specification*, RFC–1094, March 1989.

[41] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. ACM Symposium on Operating System Principles. Published as *Operating Systems Review*, **29**(5), 1995.

[42] Tripwire Open Souce 2.3.1, August 2002. `http://ftp4.sf.net/sourceforge/tripwire/tripwire-2.3.1-2.tar.gz`.

[43] K. Vaidyanathan, R. E. Harper, S. W. Hunter, and K. S. Trivedi. Analysis and implementation of software rejuvenation in cluster systems. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems. Published as *Performance Evaluation Review*, **29**(1):62–71. ACM Press, 2002.

[44] X. Zhang, L. van Doorn, T. Jaeger, R. Perez, and R. Sailer. Secure Coprocessor-based Intrusion Detection. ACM SIGOPS European Workshop. ACM, 2002.

Detecting Malicious Java Code Using Virtual Machine Auditing

Sunil Soman Chandra Krintz Giovanni Vigna

Computer Science Department
University of California, Santa Barbara
{sunils,ckrintz,vigna}@cs.ucsb.edu

Abstract

The Java Virtual Machine (JVM) is evolving as an infrastructure for the efficient execution of large-scale, network-based applications. To enable secure execution in this environment, industrial and academic efforts have implemented extensive support for verification of type-safety, authentication, and access control. However, JVMs continue to lack intrinsic support for intrusion detection.

Existing operating system auditing facilities and host-based intrusion detection systems operate at the process level, with the assumption that one application is mapped onto one process. However, in many cases, multiple Java applications are executed concurrently as threads within a single JVM process. As such, it is difficult to analyze the behavior of Java applications using the corresponding OS-level audit trail. In addition, the malicious actions of a single Java application may trigger a response that disables an entire execution environment. To overcome these limitations, we have developed a thread-level auditing facility for the Java Virtual Machine and an intrusion detection tool that uses audit data generated by this facility to detect attacks by malicious Java code. This paper describes the JVM auditing mechanisms, the intrusion detection tool, and the quantitative evaluation of their performance.

1 Introduction

Java technology [18] was initially used by web page designers to embed active content. As a result of the wide-spread success and popularity of Java, developers now use the language for implementation of a wide range of large-scale systems, e.g., robust, mobile code systems [36, 19, 33] and complex server systems [43, 32]. In these systems, multiple applications and code components are uploaded by multiple (possibly untrusted) users for concurrent execution within a single Java Virtual Machine (JVM) [38]. The portability, flexibility, and security features of Java make it an ideal technology for the implementation of systems that support the execution of mobile code.

The use of Java enables portability because programs are encoded using an architecture-independent transfer format that can be executed without modification on any platform for which a JVM has been developed. Java supports flexibility by allowing applications to be upgraded and extended at run time through the use of dynamic code loading. In addition, the Java type system and verification mechanisms provide protection against some programming errors and malicious attacks.

The key areas that must be improved to enable continued wide-spread use of Java for Internet-scale server applications are performance and security. Recent advances in Just-In-Time (JIT) compilation and optimization techniques [35, 6, 8] offer significant improvements in Java program performance. Currently, to support security in server applications, Java provides mechanisms for authentication and access control [17, 2]. However, additional mechanisms are required to detect attacks that circumvent (or attempt to circumvent) the existing protections or abuse legitimate access.

Host-based Intrusion Detection Systems (HIDSs) provide a suitable solution to this problem [3, 12]. Existing HIDSs use process-level execution events, collected by an auditing facility in the operating system,

to identify and respond to security threats and violations [13, 26, 52].

Unfortunately, since a JVM executes as a single, multi-threaded user process, we must presume that all suspicious activity reported by the auditing facility for the JVM process ID, is caused by the JVM itself. However, the culprit may be one of the *many* applications running within the JVM process. As users increasingly demand higher availability and reliability from the servers executing their applications, an intrusion response mechanism that simply terminates the JVM process that is executing malicious code becomes unacceptable. Therefore, novel auditing and intrusion detection techniques are needed, to provide finer-grained threat identification and response.

To this end, we have developed a JVM auditing facility and an intrusion detection system that detects attacks that are initiated by malicious Java code. To implement our system, We leveraged two existing technologies: a high-performance, open-source, Java Virtual Machine, called JikesRVM [1]; and an intrusion detection framework, called STAT, that we developed in prior work [50]. In this paper, we describe the JVM auditing mechanisms and the intrusion detection tool that we implemented, and provide a quantitative evaluation of the performance of our system.

In the next section, we place our work in the context of existing approaches to intrusion detection. In Section 3, we present an overview of our approach. In Section 4, we describe the JVM auditing system. Then, in Section 5, we present the intrusion detection system. Section 6 discusses the experimental evaluation of our system. Section 7 presents related work on Java security. Finally, Section 8 draws conclusions and outlines future work.

2 Extant Approaches to Intrusion Detection

Intrusion detection is performed by identifying the manifestation of a security violation in an input event stream. In host-based intrusion detection systems, the input event stream is usually represented by the audit records produced by the auditing facility of an operating system, such as the Solaris Basic Security Module [47]. Other possible input streams are system call traces and UNIX syslog messages.

The detection process can be performed according to different techniques. For example, it is possible to use statistical measures to characterize the normal behavior of users and applications [13, 23, 29]. Deviations from the established profiles are assumed to be evidence of an attack. The problem with these approaches is the difficulty to create a reliable model of the application behavior. An imprecise model may lead to both missed detections (called false negatives) and false alarms (called false positives).

Other approaches rely on the formal specification of the acceptable behavior of an application [34, 54, 52]. An execution history that does not conform to this behavior is considered malicious. The advantage of this approach is that the generation of false positives is greatly reduced. On the other hand, the generation of the specification requires access to the application source code and considerable effort, even when supported by tools. For this reason, these techniques are not in wide-spread use.

The most commonly used approach relies on models of attacks to analyze the input event stream. Systems based on this approach [20, 26, 39] are equipped with a number of attack signatures. These signatures are matched against the stream of audit data to identify the manifestation of an attack. The advantage of this approach is that it supports very effective intrusion detection and produces few false positives. In addition, signatures are not limited to modeling attacks against an application. For example, it is possible to describe attacks that represent abuses of legitimate access to the system. The disadvantage is that the signature set must be updated continuously as new ways of attacking a system are found.

In the work we present herein, we describe the design and implementation of a signature-based intrusion detection system. The system is an extension of previous work [49] in which we developed an auditing facility and an intrusion detection system for a mobile agent system, called Aglets [36]. This prior system detects malicious agent activity through the monitoring of the Aglets execution environment. In that case, the Java server application that was responsible for transferring and executing the mobile agents was instrumented to produce agent-related information.

The development of this agent monitoring system suggested a far more general approach in which the JVM itself is extended to produce the necessary information. Therefore, we developed a mechanism that collects

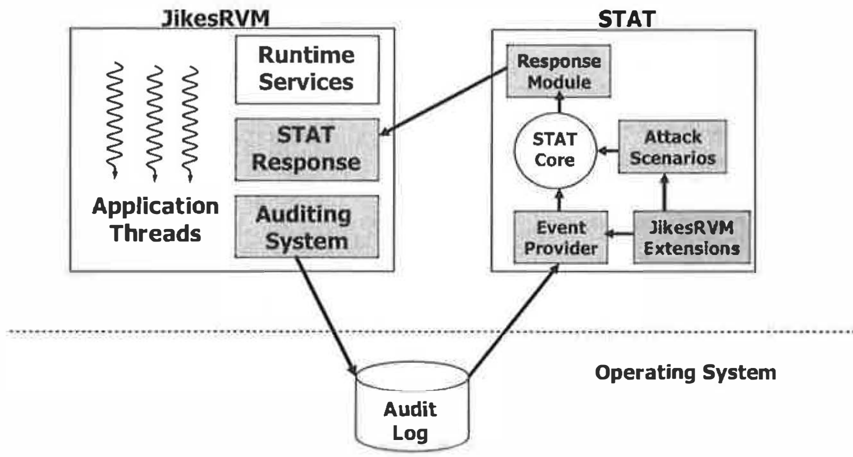


Figure 1: Architecture of the Java Virtual Machine auditing system and the STAT-based intrusion detection system

events that give information about the activity of threads within a JVM. The resulting auditing system can monitor the activity of any Java application, including various technologies supporting mobile code. We also developed an intrusion detection system that takes advantage of the finer-grained information produced by the JVM auditing system to detect attacks coming from malicious Java code.

To the best of our knowledge, no existing system performs auditing and threat detection at the Java thread level.

3 System Overview

Figure 1 shows a high-level overview of our intrusion detection architecture. The auditing system monitors executing Java threads (possibly associated with multiple, independent applications) and produces an audit log composed of records related to thread activity. The log is converted to an event stream by an event provider. The event stream is subsequently analyzed by an intrusion detection system for possible security threats or attacks. More precisely, the intrusion system compares patterns in the event stream against known attack scenarios. A match indicates that an intrusion or threat is in progress and that a response should be initiated. To do so, the intrusion detection system contacts the JVM which immediately terminates all offending threads. The JVM auditing system maps authenticated user IDs to threads so that malicious users can be identified and their threads selectively terminated.

Our system implementation couples and extends two existing frameworks: the JikesRVM, a high-performance Java virtual machine [30, 1], and STAT [51, 14], a general platform for the creation of intrusion detection sensors that we developed as part of prior work. The grayed boxes in the figure identify our extensions to these systems. In the sections that follow, we describe each of these components.

4 An Auditing System For A Java Virtual Machine

We implemented an auditing facility for the JikesRVM. The JikesRVM was designed with the goal of enabling high-performance Java server applications. As such, it represents the next generation of JVM technologies. The JikesRVM compiles Java bytecode programs into binary code (x86 or Power PC) at run time. The system operates at the method level and employs aggressive program

optimization. The JikesRVM implements extensive run-time services including, garbage collection (GC), thread scheduling, synchronization, name-space separation, and class file verification. We implemented the latter two features ourselves as part of other projects [55]. Currently, the JikesRVM enables over 25% reduction in execution time over the Sun HotSpot JVM version 1.3.1 for x86/Linux (This value refers to a number of standard benchmark programs that we also used in Section 6 of this paper to evaluate the overhead introduced by our auditing facility).

All Java threads in the JikesRVM derive from a *virtual machine thread (VM_Thread)*, which is the basic unit of program execution. These threads are multiplexed onto a *virtual processor (VM_Processor)*, which is the abstraction of an underlying operating system thread. This implementation enables the JikesRVM to perform thread scheduling, independent to that performed by the operating system.

To monitor any suspicious activity performed by applications running in the JikesRVM, we extended the virtual machine with an event logging system. Each time an application or code component is uploaded into the executing JVM, a thread is created to execute the code. The thread is assigned a unique system identifier (SID) and a user identifier (UID). The SID enables the JikesRVM auditing system to identify a specific thread when logging execution events. UIDs associate users with individual threads. Both SIDs and UIDs are inherited by every thread created by the initial thread. Strong authentication mechanisms can be used to assign UIDs to threads, however, authentication and identification are not implemented natively by the JikesRVM (we are investigating such mechanisms as part of our current research).

In this first prototype, we simply map IP addresses to user IDs; this implementation is sufficient to effectively identify malicious threads. Since each application thread that executes in the system has an associated user ID, the threads of a user can be killed without affecting other user applications or the execution environment.

Figure 2 provides a graphic description of the JVM auditing facility. The auditing facility consists of an event driver, an event queue, and an event logger. The event driver adds thread-level execution events to the event queue. The logger processes events that are contained in the queue and writes them to an external log. We next describe the event driver. The implementation of the logger component is described in Section 4.2.

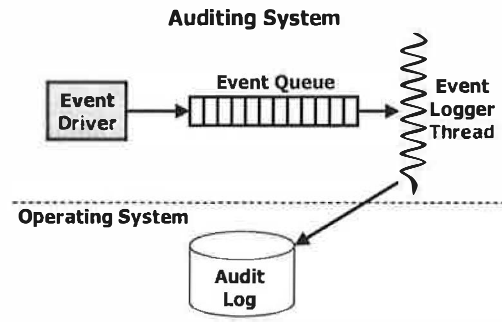


Figure 2: The JikesRVM auditing system.

4.1 Event Driver

The event driver provides an interface for inserting events in the event queue. The security-relevant operations in the JikesRVM are instrumented with calls to the event driver interface. For example, system calls invoked by the executing programs are instrumented in this way. We currently instrument operations that are of interest from a security perspective. However, any dynamic behavior can be instrumented using our system. The events we monitor in our prototype are: class events, system call events, JNI events, and thread interaction events.

4.1.1 Class Events

A class event occurs each time a thread loads a class. We instrumented the JikesRVM class loading code to record these events during execution. If a class is loaded from the network, the IP address of the code source is logged as part of the event.

We also record events associated with the creation of user-defined class loaders [37] and the classes that they load. User-defined class loaders are a powerful feature of the Java language that allows programs to define dynamically the way in which a class can be loaded and created. However, this functionality might also allow malicious applications to load classes from untrusted locations [41]. To record these events, we instrument the class loading code in user components that extend the Java ClassLoader library class [28].

4.1.2 System Call Events

System calls provide an interface through which an executing program can access operating system resources. For example, network communication, file I/O, and memory allocation are all accessed via system calls. By intercepting system calls, we are able to monitor and control such access.

The JikesRVM provides an abstraction of system calls via the VM system class. The system call routines in this class invoke the corresponding routines in the "Magic" (VM_Magic) class. The Magic class provides all the architecture-dependent, low-level functionality required by applications, e.g., raw memory access and calls to the operating system interface.

Methods in the Magic class are recognized and implemented as special system methods by both the baseline and optimizing compilers. These methods cannot be implemented by user application code. When the compiler encounters a call to a Magic method, it inlines code for the method into the caller routine. The Magic code for system calls consists of a number of calls to "wrapper routines" written in C. These routines perform the actual operating system call. To record system call events, we insert calls to the event driver interface at the VM abstraction layer (the VM system class) in the JikesRVM.

4.1.3 JNI Events

The JikesRVM implements most of the Java Native Interface (JNI) specification. The use of JNI is inherently "unsafe" since it allows user programs to call native methods, which can directly manipulate the host's file system, the process memory, and other resources. Therefore, the JNI might be used to bypass the security mechanisms of the JVM. However, the JNI offers greater control of system resources, e.g., for administration, resource accounting, and low-level device manipulation. In addition, the JNI offers the potential for improved execution performance since the code that is executed can be aggressively optimized and specialized statically for the underlying architecture, i.e., it does not require dynamic compilation. For such purposes, server systems may choose to support the use of the JNI functionality for a small, trusted subset of its users.

To monitor native method invocation, we instrument the JikesRVM JNI Compiler. The JNI compiler generates "glue code" that handles setting up the caller native method's frame for the transition from Java to C. Therefore, we inline a call to the event driver interface routine into this glue code.

4.1.4 Thread Interaction Events

In Java, an application thread can adversely affect another application thread by first obtaining a reference to the thread and then invoking one of the thread methods that could cause harmful thread-interaction, i.e., *suspend*, *interrupt*, *stop* and *kill*. In more recent versions of the Java language specification, these methods have been deprecated and it is recommended that thread interaction occur via shared variables only. However, many JVMs, including the JikesRVM, implement the direct access methods to maintain backward compatibility with legacy applications.

In Java, a thread could obtain illegal access to another thread directly, by ascending to the root thread group and recursively descending through the threads and thread groups below; or, indirectly, through the use of JNI methods that can access raw memory. Prevention of the former is straight-forward; however, the latter requires the monitoring of JNI events.

As described in the previous section, access to the JNI is granted to authorized users only. However, if the identity of a privileged user is stolen, the JNI can be used by an attacker to adversely interfere with other applications that are executing in the system. Therefore, we instrument all method invocations on thread objects that might cause interference in thread execution, including those that have been deprecated.

To record thread interaction, an instrumented method generates an event whenever the object on which it is invoked is of type Thread (*java.lang.Thread*). We modified the JikesRVM baseline and optimizing compilers so that a call to the event driver is inlined into each call of any thread method listed above, which might cause unwanted thread-interaction. In Java, an instance method is invoked as an "invokevirtual" call on an object reference. A reference to the instance resides on the stack when such a method is invoked. In our case, the object reference is a reference to the target application thread, i.e., the object on which the thread interaction method was invoked. To identify the source thread, i.e., the calling thread, we inline a call to the JikesRVM system method

VM_Thread.getCurrentThread, which gives us a reference to the thread in whose context the method call was made.

4.2 Event Logger

The event logger component runs as a JikesRVM system thread. The logger consumes events from an event queue, which the event driver interface populates. This design has the following advantages:

1. Implementing a separate event-logging thread allows us to identify and filter out events that are generated by the event logger itself (for example, system calls made by the event logger for disk I/O), by using the thread identifier.

2. Applications incur minimal delay since they do not need to wait for the disk I/O associated with the logging to complete. The application thread continues to run while the logger is processing events on the event queue.

We experienced a difficulty while instrumenting system calls. As described above, system calls are instrumented by inserting invocations to event driver interface routines, at the VM abstraction layer. The JikesRVM implementation requires that the routines in the VM class be uninterruptible. This is because an operating system call routine might directly modify the Java heap (e.g., using memcpy). Some system calls are not garbage-collection-safe (GC-safe), since they might modify the Java heap without the garbage collector's knowledge. The current JikesRVM implementation defines all system calls to be un-interruptible operations, whether or not they modify the heap. The VM stalls until the system call native code returns [25]. Hence, our event driver's event logging routines, which are called from the system call routines, cannot allocate memory to create event objects, as this might result in a garbage collection.
To solve this problem, we perform an initial static allocation of the event queue and associated event objects. Since the logger is decoupled from the driver, it can perform system calls that cause memory allocation. Therefore, the logger monitors the number of event objects in the queue, and, when a threshold is exceeded, it increases the size of the queue.

There are three drawbacks to this mechanism: (1) memory is wasted when the initial queue size is too large; (2) frequent reallocation can degrade performance if the queue size is too small; and (3) events will be missed if the queue fills before the logger thread has an opportunity to execute (and hence, increase the queue size). In our prototype, both the initial queue size and the queue allocation increment are application-dependent and can be set by the users of our system. Through empirical evaluation, using a large number of Java programs and different inputs, we determined that the queue should be doubled upon each increase and that a size of 16,000 events is sufficient to ensure that no event is ever lost. We use these values in our prototype for which we report results in Section 6.
The logger sleeps until the queue is sufficiently full. Then, the driver code wakes the logger. As with the initial queue size, this "wakeup" factor must be carefully chosen. If it is too small, the logger will be scheduled to run frequently, adversely affecting application throughput. If this factor is too large, the event driver might find the queue full and the event logger would miss events. Our empirical results for a wide range of programs and inputs indicate that this threshold should be 5/6 of queue capacity to ensure that no events are lost. The logger, when awakened, records all events that have been inserted into the queue since it was last put to sleep. It then clears the queue and goes back to sleep. Note that the event driver can continue to add events to the event queue as they are being processed by the logger. Thus, there are no extensive program interruptions due to the execution of the logger .

4.2.1 XML Encoding of Auditing Events

We encode the events that the logger consumes using an XML-based format. Encoding the events in XML supports inter-operability with other systems that may use the event stream generated by the auditing facility.
We developed a schema to encode all JikesRVM events, each of which consists of the event source, the action taken, and the result produced. Actual examples of our XML encoding are shown in Figure 3 for three JikesRVM events. The first encoding is an example of a thread interaction event in which a thread with ID 7 owned by a user with ID 23 kills a thread with ID 4 owned by the user with ID 10. The second encoding

```
<event timestamp="Wed Oct 16
        17:17:14 2002">
    <source>
        <thread id="7" uid="23" />
    </source>
    <action type="THREAD KILL"/>
        <target>
            <thread id="4" uid="10" />
        </target>
    </action>
    <result status="SUCCESS" />
    </result>
</event>
```

```
<event timestamp="Mon Nov 25
        15:30:27 2002">
    <source>
        <thread id="2" uid="17" />
    </source>
    <action type="SYSCALL sysOpen" >
        <target>
            <file name="/etc/passwd"
                mode="WRITE" />
        </target>
    </action>
    <result returncode="-1"
        status="FAILED" />
    </result>
</event>
```

```
<event timestamp="Mon Nov 25
        15:31:02 2002">
    <source>
        <thread id="8" uid="11" />
    </source>
    <action type="NET CONNECT">
        <target>
            <server remoteaddress="128.111.68.170"
                port="25640" />
        </target>
    </action>
    <result returncode="ECONNREFUSED"
        status="FAILED"/>
    </result>
</event>
```

Figure 3: XML Encoding of example JikesRVM events.

is an example of a system call event in which a thread with ID 2, owned by a user with ID 17 attempts to open the `/etc/passwd` file for writing and fails. The final example in the figure shows the encoding of a network event in which a thread, owned by user with ID 11, tries to establish a connection to a host having IP address 128.111.68.170 on port 25640 and the connection is refused.

The `source` element describes the thread (and hence, user) that initiates the operation; the thread ID (attribute `id`) and the user ID (attribute `uid`) are both attributes of the `thread` element. The `action` element describes the operation performed by the source. The `type` attribute of this action element describes the action being recorded (for example, "JNI" to denote that a native method was invoked). The `target` element describes the target to which the action is being applied. The target of an operation can be a file, a server, a method, or a thread. Of these, the latter is described by the `id` attribute; others are described using the `name` attribute. The `result` element specifies the outcome of the operation. This element has two attributes, `returncode` and `status`, which are used to record return values, e.g., from system calls, and status values, e.g., errno values.

5 Detecting Malicious Java Code

The event stream produced by the JikesRVM event logger thread is used by an intrusion detection system to identify possible threats and attacks. The intrusion detection system was developed leveraging the STAT framework [26, 50]. The STAT framework provides a generic signature-based intrusion detection engine that can be extended to match a specific environment through a well-defined process.

The first step of the extension process includes the definition of a *language extension module*. This module extends STATL [14], the domain-independent attack modeling language provided by the framework, with the event types that are specific of a particular target domain. Therefore, we developed a language extension module that defines the event types that are produced by the JikesRVM auditing facility (e.g., the `JEvent` type). By doing this, it was possible to use the JVM-specific events when writing STATL scenarios. These scenarios represent state-transition models of attacks.

An example of a scenario is shown in Figure 4. The scenario models a two-step attack in which a malicious application uses the JNI to obtain a reference to another application's thread, and then calls the "kill" method on that reference in an attempt to terminate the thread. This attack is detected by checking for a JNI method invocation (`transition_1`), followed by an attempt of the application to communicate with another user's thread (`transition_2`). This and other scenarios are presented in detail in Section 5.1.

The second step of the framework extension process is the development of an *event provider module*. This module is responsible for collecting events from the environment and translating them in the format defined in the language extension module. We developed an event provider module that reads the events contained in the audit log produced by the JikesRVM event logger and generates events in the format specified by the language extension described above. These events are matched by the STAT analysis engine against the available attack sce-

```
scenario jikesRVM_jnithread () {
    transition transition_1 (s0 → s1) nonconsuming {
        [JEvent m1]: m1.targetType == "method" &&
            m1.actionString == "JNICALL";
        {
            log("JNI method %s invoked by thread %s",
            m1.target.name, m1.source.id);
        }
    }
    transition transition_2 (s1 → s2) nonconsuming {
        [JEvent m1]: m1.targetType == "thread" &&
            m1.actionString == "THREAD KILL" &&
            m1.source.uid == m1.target.uid;
            m1.source.uid == m1.target.uid;
        {
            log("THREAD KILL by user %s's thread %s
                to user %s's thread %s",
                m1.source.id, m1.source.uid,
                m1.target.id, m1.target.uid);
        }
    }
    initial state s0 {}

    state s1 {}

    state s2 {
        log("Suspicious thread communication
            after JNI method invocation!!");
    }
}
```

Figure 4: Example attack scenario.

narios.

The third and final step of the extension process is the creation of a *response module* that can be used to react to detected attacks in a specific environment. We developed a response module that initiates an appropriate response action when an attack or threat coming from a Java application is detected. The module sends to a dedicated thread in the JikesRVM a request for a particular response action. Currently, it is possible to request the termination of a particular thread or the termination of every thread belonging to a specific user. However, the system can be easily extended to implement other types of response, such as the dynamic modification of security restrictions. The communication between the response module and the JikesRVM is secured using encryption.

In summary, by extending STAT's generic analysis engine with the modules that we developed specifically for the JikesRVM, we obtained a complete signature-based intrusion detection system that is able to detect and respond to attacks coming from malicious Java applications. The following section describes the attack scenarios that we developed for this system.

5.1 Attack Scenarios

To evaluate the efficacy with which our intrusion detection system can detect suspicious activity, we developed mobile programs that exercise different attacks and the corresponding STATL scenarios. Although we only describe four such scenarios for brevity, the system can be extended easily to detect any number of different attacks. Examples of such attacks include email forging [41] and denial of service due to continuous resource allocation [7, 41], e.g., of heap memory, files, sockets.

The four scenarios that we developed implement a range of suspicious activities including attempts to access sensitive information, suspicious inter-thread communication, ping and port scans against hosts on an internal network, and transfer of privileged information outside the network. For these scenarios, we assume a server execution model in which users connect to the JVM server and upload code that the JVM executes.

In addition, we assume that the Java type system and class file verification together enable type-safe execution [38, 37]. Our auditing facility and intrusion detection system could be bypassed if type safety is somehow violated by an attacker's program, e.g., integers are converted to references or the program counter is modified to execute at an arbitrary memory location.

5.1.1 Unauthorized Access Detection

In this attack, an application attempts to access privileged information from the host operating system, such as the password file on a UNIX host. The attack is detected by monitoring system calls that operate on file system resources. The system intercepts and records all attempts to access privileged resources. The record contains the identity of the owner of the thread and the type of access requested. Following is an example of such an

alert from the intrusion detection tool. The alert represents the attempt of an application to write to the privileged `/etc/shadow` file, which stores users' password hashes (on many UNIX systems).

```
TIME:  01/20/2003 13:44:04
ACTION: PRIVILEGED ACCESS
SOURCE UID: 0
SOURCE THREAD ID: 2
PRIVILEGED RESOURCE: /etc/shadow
ACCESS MODE: WRITE
MSG: Attempt to stat privileged file.
RESULT: FAILURE
SENSOR: jikesRVMstat@localhost
```

This scenario is an example of how the system can be used to detect malicious behavior even when an attack fails.

5.1.2 Harmful Inter-Thread Communication

A reasonably enterprising intruder might disrupt the functioning of other application threads in the system [41]. Consider the scenario in which an intruder has forged a legitimate user's identity, e.g., by stealing her user id, and has uploaded code using the stolen identity. In addition, the legitimate user is authorized to invoke native methods via the Java Native Interface (JNI). The intruder executes a native method that is designed to give her access to a thread object of another user. She then sends signals to that thread to terminate the thread. The system detects this scenario by detecting that a JNI invocation is performed followed by potentially harmful cross-thread communication between threads owned by two different users. The intrusion detection response module can communicate this information to the Java Virtual Machine, which can terminate the application uploaded by the intruder. Following is an example of such a detection alert:

```
TIME:   01/20/2003 20:44:04
ACTION: THREAD STOP after JNI Call.
SOURCE UID: 0
SOURCE THREAD ID: 2
TARGET UID: 8
TARGET THREAD ID: 1
MSG: JNI method "print" invoked by thread 2 (uid 0).
     Suspicious thread communication after
     JNI method invocation!!
RESULT: SUCCESS
SENSOR: jikesRVMstat@localhost
```

This scenario shows that in some cases malicious behavior can be detected only by using features that are internal to the JVM. Alternately, the attacker might obtain illegal access to an application thread by ascending to the thread's root thread group, and recursively descending through threads and thread groups below [41]. The intruder can then call *Thread.stop()* on the victim's thread. This single-step attack and can be handled by monitoring *Thread.stop()* calls.

5.1.3 Detecting Network Scans

With the next scenario, we show how the system can detect "bounce" attacks [24, 41] on internal servers. Using these well-known attacks, a malicious program may identify, manipulate, or discover vulnerabilities in hosts on an internal network.

In this scenario, the Java socket library is used by an attacker to perform network scans against hosts on a network behind a firewall. We assume that the JVM server system has access to these internal hosts. A malicious user uploads code that performs ping scans against internal subnets in an attempt to identify hosts that are alive. Similarly, the attacker can perform TCP and UDP scans to identify hosts with potentially vulnerable network services. To detect ping scans, we monitor the connection attempts made by an application to a range of hosts, or a range of ports on a host. Following is an example of an alert from the intrusion detection system that detects multiple connection attempts to a range of ports on host 128.111.68.170. The system identifies the thread and the user who performed the scan.

```
TIME:   01/23/2003 17:56:04
ACTION: MULTIPLE CONNECTS
SOURCE UID: 0
SOURCE THREAD ID: 2
SOURCE ADDR: 128.111.68.169
TARGET ADDR: 128.111.68.170
MSG: Connect attempts (1058) to multiple ports.
RESULT: FAILURE
SENSOR: jikesRVMstat@localhost
```

This scenario shows that it is possible to precisely identify the malicious code performing the attack using JVM-level audit data.

5.1.4 Detecting Transfer of Privileged Information

In the next scenario, we show how events that "leak" sensitive server information to the outside world [7] can be detected. The system is capable of detecting such events since we record system calls and network events.

Program	Description	Static Classes	Static Methods	Execution Time (sec)
compress	Spec JVM98 compression utility	12	44	7.11
db	Spec JVM98 database access program	3	34	16.22
jack	Spec JVM98 Java parser generator based on the Purdue Compiler Construction Tool set	56	315	4.40
java cup	LALR parser generator: A parser is created to parse simple mathematics expressions	36	385	0.38
javac	Spec JVM98 Java to bytecode compiler	176	1190	6.28
jess	Spec JVM98 expert system shell benchmark: Computes solutions to rule based puzzles	151	690	2.76
mpeg	Spec JVM98 audio file decompression tool Conforms to ISO MPEG Layer-3 spec.	55	322	5.84
mtrt	Spec JVM98 multi-threaded ray tracing implementation	26	180	3.59

Table 1: Description of the benchmarks used. The final column shows the execution performance in seconds of each benchmark executed using the JikesRVM.

An attempt to access server information, e.g., the `/etc/passwd` file, may not in itself be malicious behavior. The `/etc/passwd` file is commonly world-readable, since many UNIX utilities that run without super-user privileges depend on accessing this file to function correctly. In addition, most UNIX systems use separate shadow password files that contain the hashes of the actual passwords. However, the information contained in the */etc/passwd* file, such as account names and user information (names and addresses), might provide hints to an outside attacker, e.g., to mount a password guessing attack. As such, it may not be desirable for such information to leak out. To detect such an event, we implemented a two-step attack scenario in which an intruder accesses server information and then successfully connects to an external, untrusted machine. The alert produced for such an attempt is shown below (128.111.68.0 is our internal network).

```
TIME:   01/23/2003 17:13:58
ACTION: PRIVILEGED TRANSFER
SOURCE UID: 0
SOURCE THREAD ID: 2
INTERNAL NETWORK: 128.111.68.0
REMOTE ADDR: 128.111.43.218
PRIVILEGED RESOURCE: /etc/passwd
ACCESS TYPE: READ
MSG: Attempt to open privileged file.
     Possible transfer of privileged information
     outside internal network!
     File: /etc/passwd     Host: 128.111.43.218
RESULT: SUCCESS
SENSOR: jikesRVMstat@localhost
```

This scenario shows how the intrusion detection system can be used to associate two apparently legitimate (and authorized) operations to produce evidence of suspicious behavior.

6 Evaluation

We collected the results that we present in this section by repeatedly executing benchmark applications on the JikesRVM with and without the auditing and response components. In both cases, the execution times include the time for both dynamic (Just-In-Time) compilation of the benchmark methods and their execution. We performed the timings on an Intel Xeon 2.4GHz processor (with Hyperthreading enabled) and a Seagate 15000 rpm SCSI hard disk. The operating system is Redhat Linux 7.3, kernel version 2.4.18. The JikesRVM version we used was 2.1.1, with the FastSemispace configuration. FastSemispace implements a semispace copying garbage collector and fully optimizes all methods Just-In-Time. In addition, the entire optimizing compiler is part of the JikesRVM image at startup, i.e., class files for this subsystem have been loaded, compiled, and optimized.

The benchmark programs that we used for result generation are listed in Table 1. This set of programs is commonly used for empirical evaluation of Java-based systems and includes the Spec JVM98 [45] benchmark

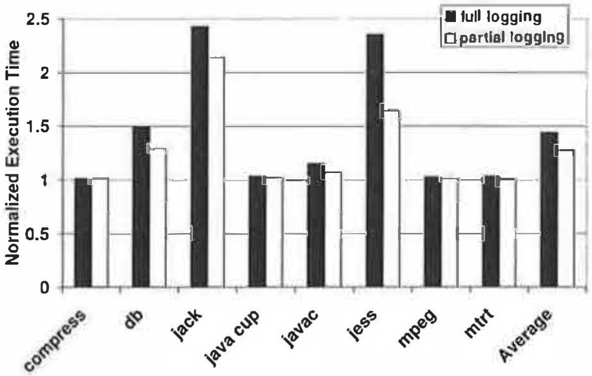


Figure 5: Execution times of our JikesRVM extension relative to the JikesRVM without auditing and response for a number of standard Java benchmarks. Full event logging implies that all system call, thread interaction, and JNI events are recorded. With partial event logging, we only log network use, file I/O, thread interaction, and JNI invocation events.

	Average Events per Second	
Program	Total Logging	Partial Logging
db	3995	956
compress	190	85
jack	12394	1894
java cup	420	209
javac	2442	400
jess	10075	3105
mpeg	1862	453
mtrt	474	240
Average	3982	918

Table 2: Event Rate (event count / execution time) for each benchmark. The first and second columns of data show the event rate for total logging and partial logging, respectively.

suite.

6.1 JikesRVM Auditing Overhead

To evaluate the efficacy of our system, we measured the overhead that the system imposes on the execution of programs for which no intrusions or threats are detected. Figure 5 shows the relative slowdown for each benchmark. The base case to which we are comparing is the JikesRVM without modification.

The delay experienced by the user application due to event logging depends on the number of events. The left-hand bar in the graph shows the impact of full logging, i.e., when all system calls, thread interaction, and JNI calls are logged. System call events generally dominate all other kinds of events, and applications with a large number of system calls incur in a substantial performance penalty. The right-hand bar, gives the performance of our system when partial logging is performed, that is when we only log events that we use in the attack scenarios presented in this paper, namely, network and file I/O, thread interaction, and JNI access. By reducing the number of events of interest, program performance improves. A user can configure our system to record any number of event types to manually adjust the trade-off between system performance and events audited.

The graph indicates that our logging introduces very little overhead in most cases (with or without full logging). For all but three benchmarks, the degradation is very small: For all benchmarks except *db*, *jack* and *jess*, performance is degraded by 5% with full logging and 2% for partial logging, on average. When we include these three benchmarks, performance is degraded by 44% for full logging and 26% for partial logging, on average.

Db, *jack* and *jess* have significant performance degradation because these benchmarks perform a large amount of file access operations, and consequently, produce a large number of events. This is clear from the average number of events per second shown in Table 2; data for both full and partial logging is shown. The relative performance degradation is greater for *jack* and *jess* than for *db* since they execute for a much shorter time (the execution times without auditing are shown in column 5 of Table 1).

Figure 6 shows the breakdown of the events generated by partial logging (100% denotes the total). These events are the number of JikesRVM system call interface routines ("wrapper routines") that are invoked by the JikesRVM Magic class. These, in turn, map onto the corresponding operating system calls: *sysStat* maps to the *stat* system call, *sysOpen* to *open*, *sysReadByte* and *sysReadBytes* to *read*, and *sysWriteByte* and *sysWriteBytes* to *write*. *Other* denotes all other non-system call events and is the only solid-colored segment. For all of the benchmarks, system call events clearly dominate other

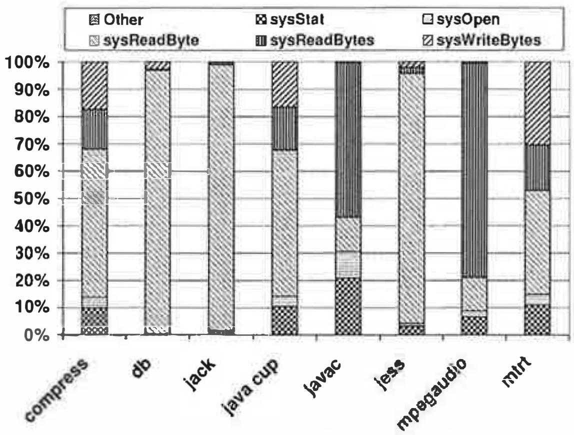


Figure 6: Breakdown of events for partial logging. These events are only those that we use for the attack scenarios that we present in this paper. Note that partial logging implies that in case of system call events, only network and file I/O events are recorded.

types of events. The number of system call events of each type varies across the benchmarks. However, file reads and writes are the most common in all cases. As part of future work, we plan to aggregate read and write system calls to reduce the overhead of instrumentation, so that events are generated only when a threshold is reached.

7 Related Work

Our work is primarily related to and complements three different areas of research. The first is extant intrusion detection research which we describe and contrast to our system in Section 2. Other related work includes alternate approaches to ensuring safe execution of mobile programs (including Java-based operating systems) and thread termination techniques.

The goal of much of the prior related research has been to develop operating systems and system management components using the Java language to enable resource management and process protection through the use of Java type-safety and load-time verification mechanisms [5, 31, 40, 22, 46, 48, 11, 21, 27, 4]. Other related work has focused on mechanisms that ensure that the execution of mobile code will not unintentionally or maliciously harm the underlying systems. Such techniques include stack inspection [16], proof-carrying code [42, 10, 9], software fault isolation [53], and code replacement [7].

The system that we propose is not an alternative to these existing approaches to program protection and system security. Instead, it offers a complementary technique (thread-level intrusion detection) that can be used to identify suspicious events or activities that may not be caught or detected by these existing approaches. For example, our system can be used within secure execution environments and Java-based operating systems to detect threads that continuously allocate memory in an attempt to cause the system to fail due to memory exhaustion or threads that perform bounce attacks on internal machines. In addition, our system is easily extensible and as such, administrators can add event detection of previously unforeseen attacks that arise but that are not handled by the underlying system.

A second area of related work is thread termination [44, 15]. In [44], the authors provide a formal specification and implementation of a technique for thread cessation called *soft termination*. Using this technique, a mobile code system can asynchronously and safely destroy the threads of a mobile program without termination of the execution environment. In our system, the response module receives messages from the STAT system when thread activity warrants its termination. The response module discontinues all threads in the system that were initiated by the user that spawned the ill-behaved thread. The module destroys non-running threads by removing them from the thread scheduling queue.

We could have implemented soft termination within the response module instead. Soft termination guarantees correct thread termination in the presence of all program and system activities, e.g., blocking system calls. As such, it is more robust and complete than our termination process. Since the JikesRVM currently only supports non-blocking system calls, we selected our simpler implementation for our initial prototype. As the JikesRVM evolves to include blocking system calls, we plan to consider the use of soft termination within our response module as part of future work.

8 Conclusions and Future work

Auditing at the Java Virtual Machine (JVM) level allows for fine-grained access to application execution events.

This information is necessary to perform effective intrusion detection and response for next-generation JVM server technologies in which multiple applications are uploaded from multiple (possibly untrusted) sites and execute concurrently within a single JVM. To this end, we developed an auditing facility for the JikesRVM and a host-based intrusion detection system that employs this audit data. As a result, attacks that exploit features internal to the JVM, such as the JNI, can now be detected. To our knowledge, this is the first system that performs auditing and intrusion detection at the thread-level within a JVM.

We also evaluated both the effectiveness of the detection process and the performance of the auditing systems. The results show that our approach introduces limited, adjustable, overhead while enabling many different attack scenarios to be detected.

Our future work will have two foci. First, we plan to extend and optimize the auditing system to handle additional events and to reduce the overhead of instrumentation. The extensions we plan include a publish-subscribe mechanism that allows intrusion detection systems to dynamically configure the auditing facility so that only the events that actually are necessary to the detection process are logged. In addition, we will use the experience that we gained with the prototype described herein, to reduce the overhead of instrumentation and audit collection within the JikesRVM.

As a second research direction, we plan to correlate traces collected at different abstraction levels to perform more effective intrusion detection. In particular, we plan to analyze the JVM-level traces with respect to application-level and OS-level traces. This integrated, multi-level approach will allow for more focused malicious code detection and a clearer evaluation of the impact of an attack on the underlying operating system.

Acknowledgments

We would like to thank the anonymous reviewers for providing extensive and useful comments on this paper.

Giovanni Vigna's work was supported by the National Science Foundation under grant CCR-0209065.

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–221, 2000.

[2] A. Anderson. Java Access Control Mechanisms. Technical report, Sun Microsystems, March 2002.

[3] J.P. Anderson. Computer Security Threat Monitoring and Surveillance. James P. Anderson Co., Fort Washington, April 1980.

[4] G. Back, P. Tullmann, L. Stoller, W. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, pages 333–346, San Diego, CA, October 2000.

[5] G. Back, P. Tullmann, L. Stoller, W. Hsieh, and J. Lepreau. Techniques for the Design of Java Operating Systems. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 197–210, San Diego, CA, June 2000.

[6] M. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Shreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Dynamic Optimizing Compiler for Java. In *ACM Java Grande Conference*, pages 129–141, June 1999.

[7] A. Chander, J. Mitchell, and I. Shin. Mobile Code Security by Java Bytecode Instrumentation. In *Proceedings of the 2001 DARPA Information Survivability Conference & Exposition (DISCEX II)*, pages 1027–1040, Anaheim, CA, June 2001.

[8] M. Cierniak, G. Lueh, and J. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 13–26, June 2000.

[9] C. Colby, K. Crary, R. Harper, P. Lee, and F. Pfenning. Automated Techniques for Provably Safe Mobile Code. In *Proceedings of the DARPA Information Survivability Conference and Exposition*, volume 1, pages 406–419, Hilton Head, SC, January 2000. IEEE Computer Society Press.

[10] C. Colby, P. Lee, G. Necula, F. Blan, M. Plesko, and K. Cline. A Certifying Compiler for Java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Program-*

ming Language Design and Implementation, pages 95–107, Vancouver, British Columbia, Canada, June 2000.

[11] G. Czajkowski and T. von Eicken. JRes: A Resource Accounting Interface for Java. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, pages 21–35, Vancouver, Canada, October 1998.

[12] H. Debar, M. Dacier, and A. Wespi. Towards a Taxonomy of Intrusion Detection Systems. *Computer Networks*, 31(8):805–822, 1999.

[13] D.E. Denning. An Intrusion Detection Model. *IEEE Transactions on Software Engineering*, 13(2):222–232, February 1987.

[14] S.T. Eckmann, G. Vigna, and R.A. Kemmerer. STATL: An Attack Language for State-based Intrusion Detection. *Journal of Computer Security*, 10(1/2):71–104, 2002.

[15] M. Flatt, R. Findler, S Krishnamurthy, and M. Felleisen. Programming Lanugages as Operating Systems (or Revenge of the Son of the Lisp Machine). In *Proceedings of the ACM International Conference on Functional Programming (ICFP'99)*, pages 138–147, Paris, France, September 1999.

[16] C. Fournet and A.D. Gordon. Stack Inspection: Theory and Variants. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 307–318, Portland, Oregon, 2002. ACM Press.

[17] L. Gong. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation.* Addison-Wesley, 1999.

[18] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[19] R.S. Gray, D. Kotz, G. Cybenko, and D. Rus. D'Agents: Security in a Multiple-language, Mobile-agent System. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 154–187. Springer-Verlag, 1998.

[20] N. Habra, B. Le Charlier, A. Mounji, and I. Mathieu. ASAX: Software Architecture and Rule-based Language for Universal Audit Trail Analysis. In *Proceedings of European Symposium on Research in Computer Security (ESORICS)*, pages 435–450, Toulouse, France, November 1992.

[21] C. Hawblitzel, C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing Multiple Protection Domains in Java. In *Proceedings of the 1998 USENIX Annual Technical Conference*, pages 259–270, New Orleans, LA, June 1998.

[22] C. Hawblitzel and T. von Eicken. Luna: A Flexible Java Protection System. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, December 2002.

[23] P. Helman and G. Liepins. Statistical Foundations of Audit Trail Analysis for the Detection of Computer Misuse. In *IEEE Transactions on Software Engineering*, volume Vol 19, No. 9, pages 886–901, 1993.

[24] Hobbit. The FTP Bounce Attack. Bugtraq, July 1995. `http://www.geocities.com/SiliconValley/1947/Ftpbounc.htm`.

[25] IBM Research. The Jikes Research Virtual Machine User's Guide. `http://www-124.ibm.com/developerworks/oss/jikesrvm/userguide/HTML/userguide.html`.

[26] K. Ilgun, R.A. Kemmerer, and P.A. Porras. State Transition Analysis: A Rule-Based Intrusion Detection System. *IEEE Transactions on Software Engineering*, 21(3):181–199, March 1995.

[27] J. Hartman and L. Peterson and A. Bavier and P. Bigot and P. Bridges and B. Montz and R. Piltz and T. Proebsting and O. Spatscheckti. Experiences Building a Communication-oriented JavaOS. *Software – Practice and Experience*, 30(10), April 2000.

[28] Java 1.2 Library API. `http://java.sun.com/products/jdk/1.2/docs/api/overview-summary.html`.

[29] H. S. Javitz and A. Valdes. The NIDES Statistical Component Description and Justification. Technical report, SRI International, Menlo Park, CA, March 1994.

[30] JikesRVM. Project Home Page. `http://www-124.ibm.com/developerworks/oss/jikesrvm`.

[31] JOS Developer Team. JOS: An Open, Portable, and Extensible Java Object Operating System. `http://cjos.sourceforge.net/archive/`.

[32] JRun. Project Home Page. `http://www.hallogram.com/jrun`.

[33] Jumping Beans. Project home page. `http://www.jumpingbeans.com`.

[34] C. Ko, M. Ruschitzka, and K. Levitt. Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-based Approach. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 175–187, Oakland, CA, May 1997.

[35] C. Krintz. Coupling On-Line and Off-Line Profile Information to Improve Program Performance. In *Proceedings of the International Symposium on Code Generation and*

Optimization (CGO03), pages 69–78, San Francisco, CA, March 2003.

[36] D.B. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets.* Addison-Wesley Longman, 1998.

[37] S. Liang and G. Bracha. Dynamic Class Loading in the Java Virtual Machine. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 36-44, Vancouver, British Columbia, Canada, 1998.

[38] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1997.

[39] U. Lindqvist and P.A. Porras. Detecting Computer and Network Misuse with the Production-Based Expert System Toolset (P-BEST). In *IEEE Symposium on Security and Privacy*, pages 146–161, Oakland, California, May 1999.

[40] M. Golm and M. Felser and C. Wawersich and J. Kleinoder. The JX Operating System. In *Proceedings of the USENIX Annual Technical Conference*, pages 45–58, Monterey, CA, June 2002.

[41] G. McGraw and E.W. Felten. *Securing Java: Getting Down to Business with Mobile Code.* John Wiley & Sons, 2nd edition, 1999.

[42] G. Necula. Proof-Carrying Code. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL97)*, pages 106–119, Paris, France, January 1997.

[43] NonStop Server for Java Software. Project Home Page. `http://nonstop.compaq.com/view.asp?IO=NSJAVAPD01`.

[44] A. Rudys and D.S. Wallach. Termination in Language-based Systems. *ACM Transactions on Information and System Security*, 5(2):138–168, 2002.

[45] SpecJVM'98 Benchmarks. `http://www.spec.org/osg/jvm98`.

[46] T. Stack, E. Eide, and J. Lepreau. Bees: A Secure, Resource-Controlled, Java-Based Execution Environment. In *Proceedings of the IEEE Conference on Open Architectures and Network Programming*, pages 97–106, San Francisco, CA, April 2003.

[47] Sun Microsystems, Inc. *Installing, Administering, and Using the Basic Security Module.* 2550 Garcia Ave., Mountain View, CA 94043, December 1991.

[48] P. Tullmann and J. Lepreau. Nested Java Processes: OS Structure for Mobile Code. In *Proceedings of the Eighth ACM SIGOPS European Workshop on Support for Composing Distributed Applications*, pages 111–117, Sintra, Portugal, 1998.

[49] G. Vigna, B. Cassel, and D. Fayram. An Intrusion Detection System for Aglets. In *Proceedings of the International Conference on Mobile Agents*, pages 64–77, Barcelona, Spain, October 2002.

[50] G. Vigna, S. Eckmann, and R. Kemmerer. The STAT Tool Suite. In *Proceedings of DISCEX 2000*, pages 1046–1055, Hilton Head, South Carolina, January 2000. IEEE Computer Society Press.

[51] G. Vigna, R.A. Kemmerer, and P. Blix. Designing a Web of Highly-Configurable Intrusion Detection Sensors. In W. Lee, L. Mè, and A. Wespi, editors, *Proceedings of the 4^{th} International Symposiun on Recent Advances in Intrusion Detection (RAID 2001)*, volume 2212 of *LNCS*, pages 69–84, Davis, CA, October 2001. Springer-Verlag.

[52] D. Wagner and D. Dean. Intrusion Detection via Static Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 156–169, Oakland, CA, May 2001. IEEE Press.

[53] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 203–216, New York, NY, USA, December 1993. ACM Press.

[54] C. Warrender, S. Forrest, and B.A. Pearlmutter. Detecting Intrusions using System Calls: Alternative Data Models. In *IEEE Symposium on Security and Privacy*, pages 133–145, 1999.

[55] L. Zhang and C. Krintz. An Investigation of Concurrent Application Execution in the JikesRVM. Technical Report TR2003-02, Dept. Computer Science, UCSB, 2003.

Static Analysis of Executables to Detect Malicious Patterns*

Mihai Christodorescu
mihai@cs.wisc.edu

Somesh Jha
jha@cs.wisc.edu

Computer Sciences Department
University of Wisconsin, Madison

Abstract

Malicious code detection is a crucial component of any defense mechanism. In this paper, we present a unique viewpoint on malicious code detection. We regard malicious code detection as an obfuscation-deobfuscation game between malicious code writers and researchers working on malicious code detection. Malicious code writers attempt to obfuscate the malicious code to subvert the malicious code detectors, such as anti-virus software. We tested the resilience of three commercial virus scanners against code-obfuscation attacks. The results were surprising: the three commercial virus scanners could be subverted by very simple obfuscation transformations! We present an architecture for detecting malicious patterns in executables that is resilient to common obfuscation transformations. Experimental results demonstrate the efficacy of our prototype tool, SAFE (a static analyzer for executables).

1 Introduction

In the interconnected world of computers, malicious code has become an omnipresent and dangerous threat. Malicious code can infiltrate hosts using a variety of methods such as attacks against known software flaws, hidden functionality in regular programs, and social engineering. Given the devastating effect malicious code has on our cyber infrastructure, identifying malicious programs is an important goal. Detecting the presence of malicious code on a given host is a crucial component of any defense mechanism.

Malicious code is usually classified [30] according to its propagation method and goal into the following categories:
• *viruses* are programs that self-replicate within a host by attaching themselves to programs and/or documents that become carriers of the malicious code;
• *worms* self-replicate across a network;
• *trojan horses* masquerade as useful programs, but contain malicious code to attack the system or leak data;
• *back doors* open the system to external entities by subverting the local security policies to allow remote access and control over a network;
• *spyware* is a useful software package that also transmits private user data to an external entity.

Combining two or more of these malicious code categories can lead to powerful attack tools. For example, a worm can contain a payload that installs a back door to allow remote access. When the worm replicates to a new system (via email or other means), the back door is installed on that system, thus providing an attacker with a quick and easy way to gain access to a large set of hosts. Staniford *et al.* have demonstrated that worms can propagate extremely quickly through a network, and thus potentially cripple the entire cyber infrastructure [43]. In a recent outbreak, the Sapphire/SQL Slammer worm reached the peak infection rate in about 10 minutes since launch, doubling every 8.5 seconds [31]. Once the back-door tool gains a large installed base, the attacker can use the compromised hosts to launch a coordinated attack, such as a distributed denial-of-service (DDoS) attack [5].

In this paper, we develop a methodology for detecting malicious patterns in executables. Although our method is general, we have initially focused our attention on viruses. A computer virus replicates itself by inserting a copy of its code (the *viral code*) into a host program. When a user executes the infected program, the virus copy runs, infects more programs, and then the original program continues to execute. To the casual user, there is no perceived difference between the clean and the in-

*This work was supported in part by the Office of Naval Research under contracts N00014-01-1-0796 and N00014-01-1-0708. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes, notwithstanding any copyright notices affixed thereon.

The views and conclusions contained herein are those of the authors, and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the above government agencies or the U.S. Government.

fected copies of a program until the virus activates its malicious payload.

The classic virus-detection techniques look for the presence of a virus-specific sequence of instructions (called a *virus signature*) inside the program: if the signature is found, it is highly probable that the program is infected. For example, the Chernobyl/CIH virus is detected by checking for the hexadecimal sequence [47]:

```
E800  0000  005B  8D4B  4251  5050
0F01  4C24  FE5B  83C3  1CFA  8B2B
```

This corresponds to the following IA-32 instruction sequence, which constitutes part of the virus body:

```
E8 00000000    call 0h
5B             pop ebx
8D 4B 42       lea ecx, [ebx + 42h]
51             push ecx
50             push eax
50             push eax
0F01 4C 24 FE  sidt [esp - 02h]
5B             pop ebx
83 C3 1C       add ebx, 1Ch
FA             cli
8B 2B          mov ebp, [ebx]
```

This classic detection approach is effective when the virus code does not change significantly over time. Detection is also easier when viruses originate from the same source code, with only minor modifications and updates. Thus, a virus signature can be common to several virus variants. For example, Chernobyl/CIH versions 1.2, 1.3, and 1.4 differ mainly in the trigger date on which the malicious code becomes active and can be effectively detected by scanning for a single signature, namely the one shown above.

The virus writers and the antivirus software developers are engaged in an *obfuscation-deobfuscation* game. Virus writers try to obfuscate the "vanilla" virus so that signatures used by the antivirus software cannot detect these "morphed" viruses. Therefore, to detect an obfuscated virus, the virus scanners first must undo the obfuscation transformations used by the virus writers. In this game, virus writers are obfuscators and researchers working on malicious code detection are deobfuscators. A method to detect malicious code should be resistant to common obfuscation transformations. This paper introduces such a method. The main contributions of this paper include:

• **The obfuscation-deobfuscation game and attacks on commercial virus scanners**
We view malicious code detection as an obfuscation-deobfuscation game between the virus writers and the researchers working to detect malicious code. Background on some common obfuscation techniques used by virus writers is given in Section 3. We also have developed an obfuscator for executables. Surprisingly, the three commercial virus scanners we considered could be easily thwarted by simple obfuscation transformations (Section 4). For example, in some cases the Norton antivirus scanner could not even detect insertions of `nop` instructions.

• **A general architecture for detecting malicious patterns in executables**
We introduce a general architecture for detecting malicious patterns in executables. An overview of the architecture and its novel features is given in Section 5. External predicates and uninterpreted symbols are two important elements in our architecture. External predicates are used to summarize results of various static analyses, such as points-to and live-range analysis. We allow these external predicates to be referred in the abstraction patterns that describe the malicious code. Moreover, we allow uninterpreted symbols in patterns, which makes the method resistant to renaming, a common obfuscation transformation. Two key components of our architecture, *the program annotator* and *the malicious code detector*, are described in Sections 6 and 7 respectively.

• **Prototype for x86 executables**
We have implemented a prototype for detecting malicious patterns in x86 executables. The tool is called a *static analyzer for executables* or *SAFE*. We have successfully tried SAFE on multiple viruses; for brevity we report on our experience with four specific viruses. Experimental results (Section 8) demonstrate the efficacy of SAFE. There are several interesting directions we intend to pursue as future work, which are summarized in Section 9.

• **Extensibility of analysis**
SAFE depends heavily on static analysis techniques. As a result, the precision of the tool directly depends on the static analysis techniques that are integrated into it. In other words, *SAFE is as good as the static analysis techniques it is built upon*. For example, if SAFE uses the result of points-to analysis, it will be able to track values across memory references. In the absence of a points-to analyzer, SAFE makes the conservative assumption that a memory reference can access any memory location (i.e., everything points to everything). We have designed SAFE so that various static analysis techniques can be readily integrated into it. Several simple static analysis techniques are already implemented in SAFE.

2 Related Work

2.1 Theoretical Discussion

The theoretical limits of malicious code detection (specifically of virus detection) have been the focus of many researchers. Cohen [10] and Chess-White [9]

showed that in general the problem of virus detection is undecidable. Similarly, several important static analysis problems are undecidable or computationally hard [28, 35].

However, the problem considered in this paper is slightly different than the one considered by Cohen [10] and Chess-White [9]. Assume that we are given a vanilla virus V which contains a malicious sequence of instructions σ. Next we are given an obfuscated version $\mathcal{O}(V)$ of the virus. The problem is to find whether there exists a sequence of instructions σ' in $\mathcal{O}(V)$ which is "semantically equivalent" to σ. A recent result by Vadhan *et al.* [3] proves that in general program obfuscation is impossible. This leads us to believe that a computationally bounded adversary will not be able to obfuscate a virus to completely hide its malicious behavior. We will further explore these theoretical issues in the future.

2.2 Other Detection Techniques

Our work is closely related to previous results on static analysis techniques for verifying security properties of software [1, 4, 8, 7, 25, 29]. In a larger context, our work is similar to existing research on software verification [2, 13]. However, there are several important differences. First, viewing malicious code detection as an obfuscation-deobfuscation game is unique. The obfuscation-deobfuscation viewpoint lead us to explore obfuscation attacks upon commercial virus scanners. Second, to our knowledge, all existing work on static analysis techniques for verifying security properties analyze source code. On the other hand, our analysis technique works on executables. In certain contexts, such as virus detection, source code is not available. Finally, we believe that using uninterpreted variables in the specification of the malicious code is unique (Section 6.2).

Currie *et al.* looked at the problem of automatically checking the equivalence of DSP routines in the context of verifying the correctness of optimizing transformations [15]. Their approach is similar to ours, but they impose a set of simplifying assumptions for their simulation tool to execute with reasonable performance. Feng and Hu took this approach one step further by using a theorem prover to determine when to unroll loops [19]. In both cases the scope of the problem is limited to VLIW or DSP code and there is limited support for user-specified analyses. Our work is applied to x86 (IA-32) assembly and can take full advantage of static analyses available to the user of our SAFE tool. Necula adopts a similar approach based on comparing a transformed code sequence against the original code sequence in the setting of verifying the correctness of the GNU C compiler [38]. Using knowledge of the transformations performed by the compiler, equivalence between the compiler input and the compiler output is proven using a simulation relation. In our work, we require no *a priori* knowledge of the obfuscation transformations performed, as it would be unrealistic to expect such information in the presence of malicious code.

We plan to enhance our framework by using the ideas from existing work on type systems for assembly code. We are currently investigating Morrisett *et al.*'s *Typed Assembly Language* [32, 33]. We apply a simple type system (Section 6) to the binaries we analyze by manually inserting the type annotations. We are unaware of a compiler that can produce Typed Assembly Language, and thus we plan to support external type annotations to enhance the power of our static analysis.

Dynamic monitoring can also be used for malicious code detection. Cohen [10] and Chess-White [9] propose a virus detection model that executes code in a sandbox. Another approach rewrites the binary to introduce checks driven by an enforceable security policy [17] (known as the *inline reference monitor* or the *IRM* approach). We believe static analysis can be used to improve the efficiency of dynamic analysis techniques, e.g., static analysis can remove redundant checks in the IRM framework. We construct our models for executables similar to the work done in specification-based monitoring [21, 46], and apply our detection algorithm in a context-insensitive fashion. Other research used context-sensitive analysis by employing push-down systems (PDSs). Analyses described in [7, 25] use the model checking algorithms for pushdown systems [18] to verify security properties of programs. The data structures used in interprocedural slicing [23], interprocedural DFA [40], and Boolean programs [2] are hierarchically structured graphs and can be translated to pushdown systems.

2.3 Other Obfuscators

While deciding on the initial obfuscation techniques to focus on, we were influenced by several existing tools. *Mistfall* (by *z0mbie*) is a library for binary obfuscation, specifically written to blend malicious code into a host program [49]. It can encrypt, morph, and blend the virus code into the host program. Our binary obfuscator is very similar to Mistfall. Unfortunately, we could not successfully morph binaries using Mistfall, so we could not perform a direct comparison between our obfuscator and Mistfall. *Burneye* (by *TESO*) is a Linux binary encapsulation tool. Burneye encrypts a binary (possibly multiple times), and packages it into a new binary with an extraction tool [45]. In this paper, we have not considered encryption based obfuscation techniques. In the future, we will incorporate encryption based obfuscation techniques into our tool, by incorporating or extending existing libraries.

3 Background on Obfuscating Viruses

To detect obfuscated viruses, antivirus software have become more complex. This section discusses some common obfuscation transformations used by virus writers and how antivirus software have historically dealt with obfuscated viruses.

A *polymorphic virus* uses multiple techniques to prevent signature matching. First, the virus code is encrypted, and only a small in-clear routine is designed to decrypt the code before running the virus. When the polymorphic virus replicates itself by infecting another program, it encrypts the virus body with a newly-generated key, and it changes the decryption routine by generating new code for it. To obfuscate the decryption routine, several transformations are applied to it. These include: `nop`-insertion, code transposition (changing the order of instructions and placing jump instructions to maintain the original semantics), and register reassignment (permuting the register allocation). These transformations effectively change the virus signature (Figure 1), inhibiting effective signature scanning by an antivirus tool.

The obfuscated code in Figure 1 will behave in the same manner as before since the `nop` instruction has no effect other than incrementing the program counter[1]. However the signature has changed. Analysis can detect simple obfuscations, like `nop`-insertion, by using regular expressions instead of fixed signatures. To catch `nop` insertions, the signature should allow for any number of `nop`s at instruction boundaries (Figure 2). In fact, most modern antivirus software use regular expressions as virus signatures.

Antivirus software deals with polymorphic viruses by performing heuristic analyses of the code (such as checking only certain program locations for virus code, as most polymorphic viruses attach themselves only at the beginning or end of the executable binary [37]), and even emulating the program in a sandbox to catch the virus in action [36]. The emulation technique is effective because at some point during the execution of the infected program, the virus body appears decrypted in main memory, ready for execution; the detection comes down to frequently scanning the in-memory image of the program for virus signatures while the program executes.

Metamorphic viruses attempt to evade heuristic detection techniques by using more complex obfuscations. When they replicate, these viruses change their code in a variety of ways, such as code transposition, substitution of equivalent instruction sequences, and register reassignment [44, 51]. Furthermore, they can "weave" the virus code into the host program, making detection by traditional heuristics almost impossible since the virus code is mixed with program code and the virus entry point is no longer at the beginning of the program (these are designated as entry point obscuring (EPO) viruses [26]).

As virus writers employ more complex obfuscation techniques, heuristic virus-detection techniques are bound to fail. Therefore, *there is need to perform a deeper analysis of malicious code based upon more sophisticated static-analysis techniques*. In other words, inspection of the code to detect malicious patterns should use structures that are closer to the semantics of the code, as purely syntactic techniques, such as regular expression matching, are no longer adequate.

3.1 The Suite of Viruses

We have analyzed multiple viruses using our tool, and discuss four of them in this paper. Descriptions of these viruses are given below.

3.1.1 Detailed Description of the Viruses

Chernobyl (CIH)

According to the Symantec Antivirus Reseach Center (SARC), *Chernobyl/CIH* is a virus that infects 32-bit Windows 95/98/NT executable files [41]. When a user executes an infected program under Windows 95/98/ME, the virus becomes resident in memory. Once the virus is resident, CIH infects other files when they are accessed. Infected files may have the same size as the original files because of CIH's unique mode of infection: the virus searches for empty, unused spaces in the file[2]. Next it breaks itself up into smaller pieces and inserts its code into these unused spaces. Chernobyl has two different payloads: the first one overwrites the hard disk with random data, starting at the beginning of the disk (sector 0) using an infinite loop. The second payload tries to cause permanent damage to the computer by corrupting the Flash BIOS.

zombie-6.b

The *z0mbie-6.b* virus includes an interesting feature – the polymorphic engine hides every piece of the virus, and the virus code is added to the infected file as a chain of differently-sized routines, making standard signature detection techniques almost useless.

f0sf0r0

The *f0sf0r0* virus uses a polymorphic engine combined with an EPO technique to hide its entry point. According to Kaspersky Labs [27], when an infected file is run and the virus code gains control, it searches for portable executable files in the system directories and infects them. While infecting, the virus encrypts itself with a polymorphic loop and writes a result to the end of the file. To gain control when the infected file is run, the virus does not modify the program's start address, but instead writes a "`jmp ⟨virus_entry⟩`" instruction into the middle of the file.

Original code	
E8 00000000	call 0h
5B	pop ebx
8D 4B 42	lea ecx, [ebx + 42h]
51	push ecx
50	push eax
50	push eax
0F01 4C 24 FE	sidt [esp - 02h]
5B	pop ebx
83 C3 1C	add ebx, 1Ch
FA	cli
8B 2B	mov ebp, [ebx]

Obfuscated code	
E8 00000000	call 0h
5B	pop ebx
8D 4B 42	lea ecx, [ebx + 45h]
90	nop
51	push ecx
50	push eax
50	push eax
90	nop
0F01 4C 24 FE	sidt [esp - 02h]
5B	pop ebx
83 C3 1C	add ebx, 1Ch
90	nop
FA	cli
8B 2B	mov ebp, [ebx]

Signature

```
E800 0000 005B 8D4B 4251 5050
0F01 4C24 FE5B 83C3 1CFA 8B2B
```

New signature

```
E800 0000 005B 8D4B 4290 5150
5090 0F01 4C24 FE5B 83C3 1C90
FA8B 2B
```

Figure 1: Original code and obfuscated code from Chernobyl/CIH, and their corresponding signatures. Newly added instructions are highlighted.

E800	0000	00(90)*	5B(90)*
8D4B	42(90)*	51(90)*	50(90)*
50(90)*	0F01	4C24	FE(90)*
5B(90)*	83C3	1C(90)*	FA(90)*
8B2B			

Figure 2: Extended signature to catch nop-insertion.

Hare

Finally, the *Hare* virus infects the bootloader sectors of floppy disks and hard drives, as well as executable programs. When the payload is triggered, the virus overwrites random sectors on the hard disk, making the data inaccessible. The virus spreads by polymorphically changing its decryption routine and encrypting its main body.

The Hare and Chernobyl/CIH viruses are well known in the antivirus community, with their presence in the wild peaking in 1996 and 1998, respectively. In spite of this, we discovered that *current commercial virus scanners could not detect slightly obfuscated versions of these viruses.*

4 Obfuscation Attacks on Commercial Virus Scanners

We tested three commercial virus scanners against several common obfuscation transformations. To test the resilience of commercial virus scanners to common obfuscation transformations, we have developed an obfuscator for binaries. Our obfuscator supports four common obfuscation transformations: dead-code insertion, code transposition, register reassignment, and instruction substitution. While there are other generic obfuscation techniques [11, 12], those described here seem to be preferred by malicious code writers, possibly because implementing them is easy and they add little to the memory footprint.

4.1 Common Obfuscation Transformations

4.1.1 Dead-Code Insertion

Also known as *trash insertion*, dead-code insertion adds code to a program without modifying its behavior. Inserting a sequence of nop instructions is the simplest example. More interesting obfuscations involve constructing challenging code sequences that modify the program state, only to restore it immediately.

Some code sequences are designed to fool antivirus software that solely rely on signature matching as their detection mechanism. Other code sequences are complicated enough to make automatic analysis very time-consuming, if not impossible. For example, passing values through memory rather than registers or the stack requires accurate pointer analysis to recover values. The example shown in Figure 3 should clarify this. The code marked by (*) can be easily eliminated by automated analysis. On the other hand, the second and third insertions, marked by (**), do cancel out but the analysis is more complex. Our obfuscator supports dead-code insertion.

Not all dead-code sequence can be detected and eliminated, as this problem reduces to program equivalence (i.e., *is this code sequence equivalent to an empty program?*), which is undecidable. We believe that many common dead-code sequences can be detected and eliminated with acceptable performance. To quote the docu-

mentation of the RPME virus permutation engine [50],

> [T]rash [does not make the] program more complex [...]. If [the] detecting algorithm will be written such as I think, then there is no difference between NOP and more complex trash.

Our detection tool, SAFE, identifies several kinds of such dead-code segments.

4.1.2 Code Transposition

Code transposition shuffles the instructions so that the order in the binary image is different from the execution order, or from the order of instructions assumed in the signature used by the antivirus software. To achieve the first variation, we randomly reorder the instructions and insert unconditional branches or *jumps* to restore the original control-flow. The second variation swaps instructions if they are not interdependent, similar to compiler code generation, but with the different goal of randomizing the instruction stream.

The two versions of this obfuscation technique differ in their complexity. The code transposition technique based upon unconditional branches is relatively easy to implement. The second technique that interchanges independent instructions is more complicated because the independence of instructions must be ascertained. On the analysis side, code transposition can complicate matters only for a human. Most automatic analysis tools (including ours) use an intermediate representation, such as the control flow graph (CFG) or the program dependence graph (PDG) [23], that is not sensitive to superfluous changes in control flow. Note that an optimizer acts as a deobfuscator in this case by finding the unnecessary unconditional branches and removing them from the program code. Currently, our obfuscator supports only code transposition based upon inserting unconditional branches.

4.1.3 Register Reassignment

The register reassignment transformation replaces usage of one register with another in a specific live range. This technique exchanges register names and has no other effect on program behavior. For example, if register `ebx` is dead throughout a given live range of the register `eax`, it can replace `eax` in that live range. In certain cases, register reassignment requires insertion of prologue and epilogue code around the live range to restore the state of various registers. Our binary obfuscator supports this code transformation.

The purpose of this transformation is to subvert the antivirus software analyses that rely upon signature-matching. There is no real obfuscatory value gained in this process. Conceptually, the deobfuscation challenge is equally complex before or after the register reassignment.

4.1.4 Instruction Substitution

This obfuscation technique uses a dictionary of equivalent instruction sequences to replace one instruction sequence with another. Since this transformation relies upon human knowledge of equivalent instructions, it poses the toughest challenge for automatic detection of malicious code. The IA-32 instruction set is especially rich, and provides several ways of performing the same operation. Coupled with several architecturally ambivalent features (e.g., a memory-based stack that can be accessed both as a stack using dedicated instructions and as a memory area using standard memory operations), the IA-32 assembly language provides ample opportunity for instruction substitution.

To handle obfuscation based upon instruction substitution, an analysis tool must maintain a dictionary of equivalent instruction sequences, similar to the dictionary used to generate them. This is not a comprehensive solution, but it can cope with the common cases. In the case of IA-32, the problem can be slightly simplified by using a simple intermediate language that "unwinds" the complex operations corresponding to each IA-32 instruction. In some cases, a theorem prover such as Simplify [16] or PVS [39] can also be used to prove that two sequences of instructions are equivalent.

4.2 Testing Commercial Antivirus Tools

We tested three commercial virus scanners using obfuscated versions of the four viruses described earlier. The results were quite surprising: *a combination of* `nop`*-insertion and code transposition was enough to create obfuscated versions of the viruses that the commercial virus scanners could not detect.* Moreover, the Norton antivirus software could not detect an obfuscated version of the Chernobyl virus using just `nop`-insertions. SAFE was resistant to the two obfuscation transformations. The results are summarized in Table 1. A ✓ indicates that the antivirus software detected the virus. A ✗ means that the software did not detect the virus. Note that unobfuscated versions of all four viruses were detected by all the tools.

5 Architecture

This section gives an overview of the architecture of *SAFE* (Figure 4). Subsequent sections provide detailed descriptions of the major components of SAFE.

To detect malicious patterns in executables, we build an abstract representation of the malicious code (here a virus). The abstract representation is the "generalization" of the malicious code, e.g., it incorporates obfuscation transformations, such as superfluous changes

Original code

```
call 0h
pop ebx
lea ecx, [ebx+42h]
push ecx
push eax
push eax
sidt [esp - 02h]
pop ebx
add ebx, 1Ch
cli
mov ebp, [ebx]
```

Code obfuscated through dead-code insertion

```
call 0h
pop ebx
lea ecx, [ebx+42h]
nop                     (*)
nop                     (*)
push ecx
push eax
inc eax                 (**)
push eax
dec [esp - 0h]          (**)
dec eax                 (**)
sidt [esp - 02h]
pop ebx
add ebx, 1Ch
cli
mov ebp, [ebx]
```

Code obfuscated through code transposition

```
     call 0h
     pop ebx
     jmp S2
S3:  push eax
     push eax
     sidt [esp - 02h]
     jmp S4
     add ebx, 1Ch
     jmp S6
S2:  lea ecx, [ebx+42h ]
     push ecx
     jmp S3
S4:  pop ebx
     cli
     jmp S5
S5:  mov ebp, [ebx]
```

Code obfuscated through instruction substitution

```
call 0h
pop ebx
lea ecx, [ebx+42h]
sub esp, 03h
sidt [esp - 02h]
add [esp], 1Ch
mov ebx, [esp]
inc esp
cli
mov ebp, [ebx]
```

Figure 3: Examples of obfuscation through dead-code insertion, code transposition, and instruction substitution. Newly added instructions are highlighted.

		Norton® Antivirus 7.0	McAfee® VirusScan 6.01	Command® Antivirus 4.61.2	SAFE
Chernobyl	original	✓	✓	✓	✓
	obfuscated	×[1]	×[1,2]	×[1,2]	✓
z0mbie-6.b	original	✓	✓	✓	✓
	obfuscated	×[1,2]	×[1,2]	×[1,2]	✓
f0sf0r0	original	✓	✓	✓	✓
	obfuscated	×[1,2]	×[1,2]	×[1,2]	✓
Hare	original	✓	✓	✓	✓
	obfuscated	×[1,2]	×[1,2]	×[1,2]	✓

Obfuscations considered: [1] = nop-insertion (a form of dead-code insertion)
[2] = code transposition

Table 1: Results of testing various virus scanners on obfuscated viruses.

in control flow and register reassignments. Similarly, one must construct an abstract representation of the executable in which we are trying to find a malicious pattern. Once the generalization of the malicious code and the abstract representation of the executable are created, we can then detect the malicious code in the executable. We now describe each component of SAFE.

Generalizing the malicious code: Building the malicious code automaton

The malicious code is generalized into an automaton with uninterpreted symbols. Uninterpreted symbols (Section 6.2) provide a generic way of representing data dependencies between variables without specifically referring to the storage location of each variable.

Pattern-definition loader

This component takes a library of *abstraction patterns* and creates an internal representation. These abstraction patterns are used as alphabet symbols by the malicious code automaton.

The executable loader

This component transforms the executable into an internal representation, here the collection of control flow graphs (CFGs), one for each program procedure. The executable loader (Figure 5) uses two off-the-shelf components, *IDA Pro* and *CodeSurfer*. IDA Pro (by DataRescue [42]) is a commercial interactive disassembler. *CodeSurfer* (by GrammaTech, Inc. [24]) is a program-understanding tool that performs a variety of static analyses. CodeSurfer provides an API for access to various structures, such as the CFGs and the call graph, and to results of a variety of static analyses, such as points-to analysis. In collaboration with GrammaTech, we have developed a connector that transforms IDA Pro internal structures into an intermediate form that CodeSurfer can parse.

The annotator

This component inputs a CFG from the executable and the set of abstraction patterns and produces an annotated CFG, the abstract representation of a program procedure. The annotated CFG includes information that indicates where a specific abstraction pattern was found in the executable. The annotator runs for each procedure in the program, transforming each CFG. Section 6 describes the annotator in detail.

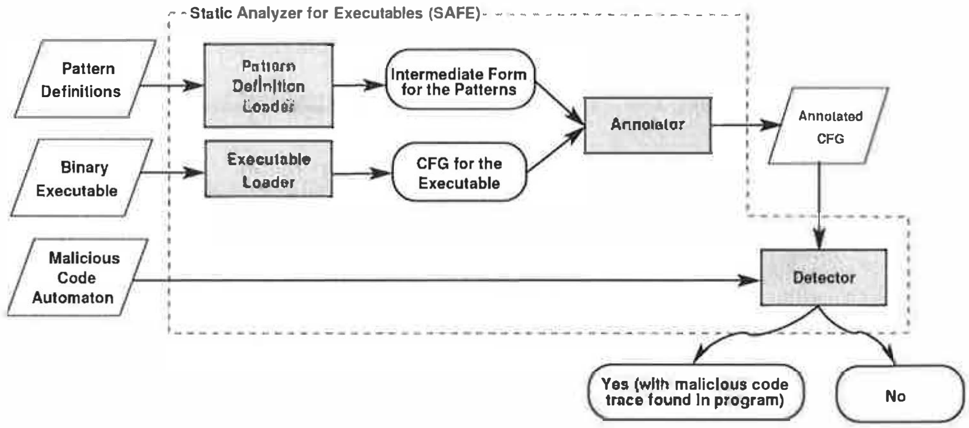


Figure 4: Architecture of the static analyzer for executables (SAFE).

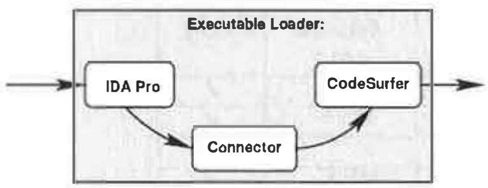


Figure 5: Implementation of executable loader module.

The detector

This component computes whether the malicious code (represented by the malicious code automaton) appears in the abstract representation of the executable (created by the annotator). This component uses an algorithm based upon language containment and unification. Details can be found in Section 7.

Throughout the rest of the paper, the malicious code fragment shown in Figure 6 is used as a running example. This code fragment was extracted from the Chernobyl virus version 1.4.

Original code

```
  WVCTF:
              mov     eax, dr1
              mov     ebx, [eax+10h]
              mov     edi, [eax]
LOWVCTF:
              pop     ecx
              jecxz   SFMM
              mov     esi, ecx
              mov     eax, 0d601h
              pop     edx
              pop     ecx
              call    edi
              jmp     LOWVCTF
   SFMM:
              pop     ebx
              pop     eax
              stc
              pushf
```

Figure 6: Original code fragment from Chernobyl virus version 1.4.

To obtain the obfuscated code fragment depicted (Figure 7), we applied the following obfuscation transformations: dead-code insertion, code transposition, and register reassignment. Incidentally, the three commercial antivirus software (Norton, McAfee, and Command) detected the original code fragment shown. However, the obfuscated version was not detected by any of the three commercial antivirus software.

6 Program Annotator

This section describes the program annotator in detail and the data structures and static analysis concepts used in the detection algorithm. The program annotator inputs the CFG of the executable and a set of abstraction patterns and outputs an annotated CFG. The annotated CFG associates with each node n in the CFG a set of patterns that match the program at the point corresponding to the node n. The precise syntax for an abstraction pattern and the semantics of matching are provided later in the section.

Figure 8 shows the CFG and a simple annotated CFG corresponding to the obfuscated code from Figure 7. Note that one node in the annotated CFG can correspond to several nodes in the original CFG. For example, the nodes annotated with "IrrelevantInstr" corresponds to one or more nop instructions.

The annotations that appear in Figure 8 seem intuitive, but formulating them within a static-analysis framework requires formal definitions. We enhance the SAFE framework with a type system for x86 based on the typestate system described in [48]. However, other type systems designed for assembly languages, such as *Typed Assembly Language* [32, 33], could be used in the SAFE framework. Definitions, patterns, and the matching procedure are described in Sections 6.1, 6.2 and 6.3 respectively.

Obfuscated code

```
WVCTF:
            mov     eax, dr1
            jmp     Loc1
   Loc2:
            mov     edi, [eax]
LOWVCTF:
            pop     ecx
            jecxz   SFMM
            nop
            mov     esi, ecx
            nop
            nop
            mov     eax, 0d601h
            jmp     Loc3
   Loc1:
            mov     ebx, [eax+10h]
            jmp     Loc2
   Loc3:
            pop     edx
            pop     ecx
            nop
            call    edi
            jmp     LOWVCTF
   SFMM:
            pop     ebx
            pop     eax
            push    eax
            pop     eax
            stc
            pushf
```

Figure 7: Obfuscated version based upon code in Figure 6.

6.1 Basic Definitions

This section provides the formal definitions used in the rest of the paper.

Program Points

An *instruction* I is a function application, $I : \tau_1 \times \cdots \times \tau_k \to \tau$. While the type system does not preclude higher-order functions or function composition, it is important to note that most assembly languages (including x86) do not support these concepts. A *program* P is a sequence of instructions $\langle I_1, \ldots, I_N \rangle$. During program execution, the instructions are processed in the sequential order they appear in the program, with the exception of control-flow instructions that can change the sequential execution order. The index of the instruction in the program sequence is called a *program point* (or *program counter*), denoted by the function $pc : \{I_1, \ldots, I_N\} \to [1, \ldots, N]$, and defined as $pc(I_j) \stackrel{def}{=} j,\ \forall\ 1 \le j \le N$. The set of all program points for program P is $ProgramPoints(P) \stackrel{def}{=} \{1, \ldots, N\}$. The pc function provides a total ordering over the set of program instructions.

Control Flow Graph

A *basic block* B is a maximal sequence of instructions $\langle I_l, \ldots, I_m \rangle$ that contains at most one control-flow instruction, which must appear at the end. Thus, the execution within a basic block is by definition sequential. Let V be the set of basic blocks for a program P, and $E \subseteq V \times V \times \{T, F\}$ be the set of control flow transitions between basic blocks. Each edge is marked with either T or F corresponding to the condition ($true$ or $false$) on which that edge is followed. Unconditional jumps have outgoing edges always marked with T. The directed graph $CFG(P) = \langle V, E \rangle$ is called the *control flow graph.*

Predicates

Predicates are the mechanism by which we incorporate results of various static analyses such as live range and points-to analysis. These predicates can be used in the definition of abstraction patterns. Table 2 lists predicates that are currently available in our system. For example, code between two program points p_1 and p_2 can be verified as dead-code (Section 4.1.1) by checking that for every variable m that is live in the program range $[p_1, p_2]$, its value at point p_2 is the same as its value at point p_1. The change in m's value between two program points p_1 and p_2 is denoted by $Delta(m, p_1, p_2)$ and can be implemented using polyhedral analysis [14].

Explanations of the static analysis predicates shown in Table 2 are standard and can be found in a compiler textbook (such as [34]).

Instructions and Data Types

The type constructors build upon simple integer types (listed below as the *ground* class of types), and allow for array types (with two variations: the pointer-to-start-of-array type and the pointer-to-middle-of-array type), structures and unions, pointers, and functions. Two special types $\bot(n)$ and $\top(n)$ complete the type system lattice. $\bot(n)$ and $\top(n)$ represent types that are stored on n bits, with $\bot(n)$ being the least specific ("any") type and $\top(n)$ being the most specific type. Table 3 describes the constructors allowed in our type system.

The type $\mu(l, \tau, i)$ represents the type of a field member of a structure. The field has a type τ (independent of the types of all other fields in the same structure), an offset i that uniquely determines the location of the field within the structure, and a label l that identifies the field within the structure (in some cases this label might be undefined).

Physical subtyping takes into account the layout of values in memory [6, 48]. If a type τ is a *physical subtype* of τ' (denoted it by $\tau \le \tau'$), then the memory layout of a value of type τ' is a prefix of the memory layout of a value of type τ. We will not describe the rules of physical subtyping here as we refer the reader to Xu's thesis [48] for a detailed account of the typestate system

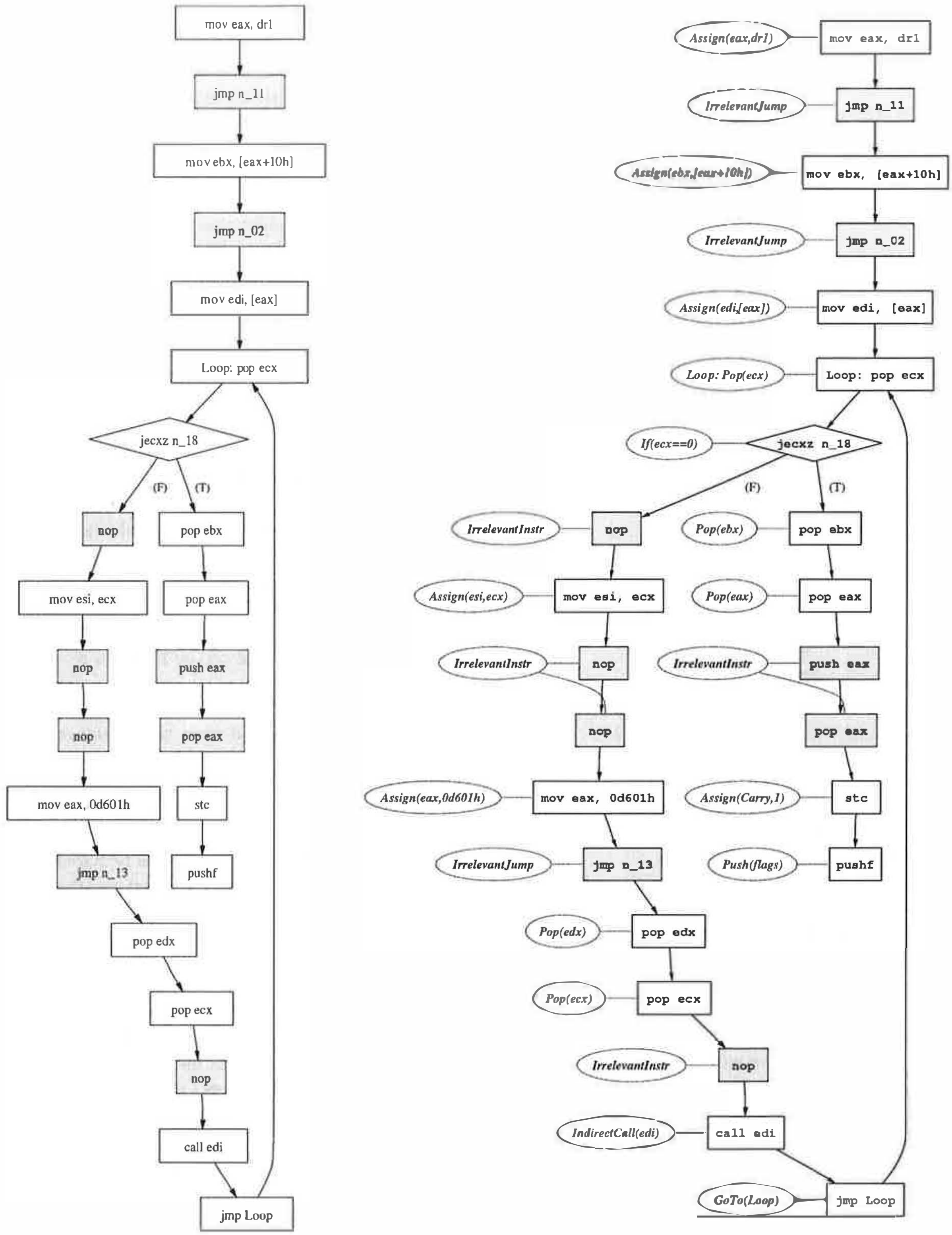


Figure 8: Control flow graph of obfuscated code fragment, and annotations.

$Dominators(B)$	the set of basic blocks that dominate the basic block B
$PostDominators(B)$	the set of basic blocks that are dominated by the basic block B
$Pred(B)$	the set of basic blocks that immediately precede B
$Succ(B)$	the set of basic blocks that immediately follow B
$First(B)$	the first instruction of the basic block B
$Last(B)$	the last instruction of the basic block B
$Previous(I)$	$\begin{cases} \bigcup_{B' \in Pred(B_I)} Last(B') & \text{if } I = First(B_I) \\ I' & \text{if } B_I = \langle \ldots, I', I, \ldots \rangle \end{cases}$
$Next(I)$	$\begin{cases} \bigcup_{B' \in Succ(B_I)} First(B') & \text{if } I = Last(B_I) \\ I' & \text{if } B_I = \langle \ldots, I, I', \ldots \rangle \end{cases}$
$Kills(p, a)$	*true* if the instruction at program point p kills variable a
$Uses(p, a)$	*true* if the instruction at program point p uses variable a
$Alias(p, x, y)$	*true* if variable x is an alias for y at program point p
$LiveRangeStart(p, a)$	the set of program points that start the a's live range that includes p
$LiveRangeEnd(p, a)$	the set of program points that end the a's live range that includes p
$Delta(p, m, n)$	the difference between integer variables m and n at program point p
$Delta(m, p_1, p_2)$	the change in m's value between program points p_1 and p_2
$PointsTo(p, x, a)$	*true* if variable x points to location of a at program point p

Table 2: Examples of static analysis predicates.

τ	::	ground	*Ground types*
	|	τ`[n]`	*Pointer to the base of an array of type* τ *and of size* n
	|	τ`(n)`	*Pointer into the middle of an array of type* τ *and of size* n
	|	τ `ptr`	*Pointer to* τ
	|	$s\{\mu_1, \ldots, \mu_k\}$	*Structure (product of types of* μ_i*)*
	|	$u\{\mu_1, \ldots, \mu_k\}$	*Union*
	|	$\tau_1 \times \cdots \times \tau_k \to \tau$	*Function*
	|	$\top$`(n)`	*Top type of* n *bits*
	|	$\bot$`(n)`	*Bottom type of* n *bits (type "any" of* n *bits)*
μ	::	(l, τ, i)	*Member labeled* l *of type* τ *at offset* i
ground	::	`int(`g`:`s`:`v`)` | `uint(`g`:`s`:`v`)` | ...	

Table 3: A simple type system.

(including subtyping rules).

The type `int(`g`:`s`:`v`)` represents a signed integer, and it covers a wide variety of values within storage locations. It is parametrized using three parameters as follows: g represents the number of highest bits that are ignored, s is the number of middle bits that represent the sign, and v is the number of lowest bits that represent the value. Thus the type `int(`g`:`s`:`v`)` uses a total of $g + s + v$ bits.

$$\underbrace{d_{g+s+v} \ldots d_{s+v+1}}_{\text{ignored}} \underbrace{d_{s+v} \ldots d_{v+1}}_{\text{sign}} \underbrace{d_v \ldots d_1}_{\text{value}}$$

The type `uint(`g`:`s`:`v`)` represents an unsigned integer, and it is just a variation of `int(`g`:`s`:`v`)`, with the middle s sign bits always set to zero.

The notation `int(`g`:`s`:`v`)` allows for the separation of the data and storage location type. In most assembly languages, it is possible to use a storage location larger than that required by the data type stored in it. For example, if a byte is stored right-aligned in a (32-bit) word, its associated type is `int(24:1:7)`. This means that an instruction such as *xor on least significant byte within 32-bit word* will preserve the leftmost 24 bits of the 32-bit word, even though the instruction addresses the memory on 32-bit word boundary.

This separation between data and storage location raises the issue of alignment information, i.e., most computer systems require or prefer data to be at a memory address aligned to the data size. For example, 32-bit integers should be aligned on 4-byte boundaries, with the drawback that accessing an unaligned 32-bit integer leads to either a slowdown (due to several aligned memory accesses) or an exception that requires handling in software. Presently, we do not use alignment information as it does not seem to provide a significant covert way of changing the program flow.

Figure 9 shows the types for operands in a section of code from the Chernobyl/CIH virus. Table 4 illustrates the type system for Intel IA-32 architecture. There are other IA-32 data types that are not covered in Table 4, including bit strings, byte strings, 64- and 128-bit packed SIMD types, and BCD and packed BCD formats. The

Code	Type
`call 0h`	
`pop ebx`	`ebx: ⊥(32)`
`lea ecx, [ebx + 42h]`	`ecx: ⊥(32), ebx: ptr ⊥(32)`
`push ecx`	`ecx: ⊥(32)`
`push eax`	`eax: ⊥(32)`
`push eax`	`eax: ⊥(32)`
`sidt [esp - 02h]`	
`pop ebx`	`eax: ⊥(32)`
`add ebx, 1Ch`	`ebx: int(0:1:31)`
`cli`	
`mov ebp, [ebx]`	`ebp: ⊥(32), ebx: ptr ⊥(32)`

Figure 9: Inferred types from Chernobyl/CIH virus code.

IA-32 logical address is a combination of a 16-bit segment selector and a 32-bit segment offset, thus its type is the cross product of a 16-bit unsigned integer and a 32-bit pointer.

6.2 Abstraction Patterns

An abstraction pattern Γ is a 3-tuple (V, O, C), where V is a list of typed variables, O is a sequence of instructions, and C is a boolean expression combining one or more static analysis predicates over program points. Formally, a pattern $\Gamma = (V, O, C)$ is a 3-tuple defined as follows:

$$\begin{aligned} V &= \{ x_1 : \tau_1, \ldots, x_k : \tau_k \} \\ O &= \langle\, I(v_1, \ldots, v_m) \mid I : \tau_1 \times \cdots \times \tau_m \to \tau \,\rangle \\ C &= \text{boolean expression involving static analysis predicates and logical operators} \end{aligned}$$

An instruction from the sequence O has a number of arguments $(v_i)_{i \geq 0}$, where each argument is either a literal value or a free variable x_j. We write $\Gamma(x_1 : \tau_1, \ldots, x_k : \tau_k)$ to denote the pattern $\Gamma = (V, O, C)$ with free variables $x_1, \ldots, x_k$. An example of a pattern is shown below.

$$\begin{aligned} \Gamma(\, X : int(0:1:31)\,) = \\ (\ \ \{ X : int(0:1:31) \}, \\ \langle\, p_1 : \text{"pop } X\text{"}, \\ p_2 : \text{"add } X, \texttt{03AFh}\text{"}\ \rangle, \\ p_1 \in LiveRangeStart(p_2, X)\ ) \end{aligned}$$

This pattern represents two instructions that pop a register X off the stack and then add a constant value to it (`0x03AF`). Note the use of uninterpreted symbol X in the pattern. Use of the uninterpreted symbols in a pattern allows it to match multiple sequences of instructions, e.g., the patterns shown above matches any instantiation of the pattern where X is assigned a specific register. The type $int(0:1:31)$ of X represents an integer with 31 bits of storage and one sign bit.

We define a *binding* $\mathcal{B}$ as a set of pairs [variable v, value x]. Formally, a binding $\mathcal{B}$ is defined as $\{ [x, v] \mid x \in V,\ x : \tau,\ v : \tau',\ \tau \preceq \tau' \}$. If a pair $[x, v]$ occurs in a binding $\mathcal{B}$, then we write $\mathcal{B}(x) = v$. Two bindings $\mathcal{B}_1$ and $\mathcal{B}_2$ are said to be *compatible* if they do not bind the same variable to different values:

$$\begin{aligned} &Compatible(\mathcal{B}_1, \mathcal{B}_2) \stackrel{def}{=} \\ &\quad \forall\, x \in V.(\ [x, y_1] \in \mathcal{B}_1 \ \wedge\ [x, y_2] \in \mathcal{B}_2\ ) \\ &\qquad \Rightarrow (y_1 = y_2) \end{aligned}$$

The *union of two compatible bindings* $\mathcal{B}_1$ and $\mathcal{B}_2$ includes all the pairs from both bindings. For incompatible bindings, the union operation returns an empty binding.

$$\mathcal{B}_1 \cup \mathcal{B}_2 \stackrel{def}{=} \begin{cases} \{\ [x, v_x] : [x, v_x] \in \mathcal{B}_1 \vee [x, v_x] \in \mathcal{B}_2\ \} & \text{if } Compatible(\mathcal{B}_1, \mathcal{B}_2) \\ \emptyset & \text{if } \neg\ Compatible(\mathcal{B}_1, \mathcal{B}_2) \end{cases}$$

When matching an abstraction pattern against a sequence of instructions, we use unification to bind the free variables of Γ to actual values. The function

$$Unify\ (\ \langle \ldots, op_i(x_{i,1}, \ldots, x_{i,n_i}), \ldots \rangle_{1 \leq i \leq m},\ \Gamma)$$

returns a "most general" binding $\mathcal{B}$ if the instruction sequence $\langle \ldots, op_i(x_{i,1}, \ldots, x_{i,n_i}), \ldots \rangle_{1 \leq i \leq m}$ can be unified with the sequence of instructions O specified in the pattern Γ. If the two instruction sequences cannot be unified, *Unify* returns *false*. Definitions and algorithms related to unification are standard and can be found in [20].[3]

6.3 Annotator Operation

The annotator associates a set of matching patterns with each node in the CFG. The annotated CFG of a program procedure P with respect to a set of patterns Σ is denoted by P_Σ. Assume that a node n in the CFG corresponds to the program point p and the instruction at p is I_p. The annotator attempts to match the (possibly interprocedural) instruction sequence $S(n) = \langle \ldots, Previous^2(I_p), Previous(I_p), I_p \rangle$ with the patterns in the set $\Sigma = \{\Gamma_1, \ldots, \Gamma_m\}$. The CFG node n is then labeled with the list of pairs of patterns and bindings that satisfy the following condition:

$$\begin{aligned} Annotation(n) = \{\ [\Gamma, \mathcal{B}]\ : \Gamma \in \{\Gamma_1, \ldots, \Gamma_m\} \wedge \\ \mathcal{B} = Unify(S(n), \Gamma)\ \} \end{aligned}$$

If $Unify(S(n), \Gamma)$ returns $false$ (because unification is not possible), then the node n is not annotated with $[\Gamma, \mathcal{B}]$. Note that a pattern Γ might appear several times (albeit with different bindings) in $Annotation(n)$. However, the pair $[\Gamma, \mathcal{B}]$ is unique in the annotation set of a given node.

IA-32 Datatype	*Type Expression*
`byte unsigned int`	`uint`(0:0:8)
`word unsigned int`	`uint`(0:0:16)
`doubleword unsigned int`	`uint`(0:0:32)
`quadword unsigned int`	`uint`(0:0:64)
`double quadword unsigned int`	`uint`(0:0:128)
`byte signed int`	`int`(0:1:7)
`word signed int`	`int`(0:1:15)
`doubleword signed int`	`int`(0:1:31)
`quadword signed int`	`int`(0:1:63)
`double quadword signed int`	`int`(0:1:127)
`single precision float`	`float`(0:1:31)
`double precision float`	`float`(0:1:63)
`double extended precision float`	`float`(0:1:79)
`near pointer`	$\perp$(32)
`far pointer (logical address)`	`uint`(0:0:16) $\times$ `uint`(0:0:32) $\rightarrow \perp$(48)
`eax, ebx, ecx, edx`	$\perp$(32)
`esi, edi, ebp, esp`	$\perp$(32)
`eip`	`int`(0:1:31)
`cs, ds, ss, es, fs, gs`	$\perp$(16)
`ax, bx, cx, dx`	$\perp$(16)
`al, bl, cl, dl`	$\perp$(8)
`ah, bh, ch, dh`	$\perp$(8)

Table 4: IA-32 datatypes and their corresponding expression in the type system from Table 3.

7 Detector

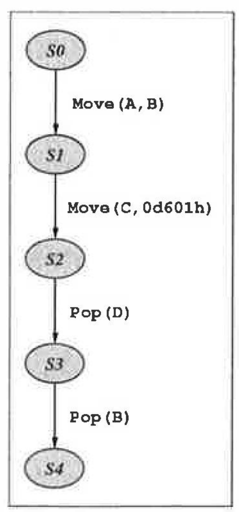
The detector takes as its inputs an annotated CFG for an executable program procedure and a malicious code automaton. If the malicious pattern described by the malicious code automaton is also found in the annotated CFG, the detector returns the sequence of instructions exhibiting the pattern. The detector returns *no* if the malicious pattern cannot be found in the annotated CFG.

7.1 The Malicious-Code Automaton

Intuitively, the malicious code automaton is a generalization of the vanilla virus, i.e., the malicious code automaton also represents obfuscated strains of the virus. Formally, a *malicious code automaton* (or *MCA*) $\mathcal{A}$ is a 6-tuple $(V, \Sigma, S, \delta, S_0, F)$, where

- $V = \{v_1 : \tau_1, \ldots, v_k : \tau_k\}$ is a *set of typed variables*,
- $\Sigma = \{\Gamma_1, \ldots, \Gamma_n\}$ is a *finite alphabet* of patterns parametrized by variables from V, for $1 \leq i \leq n$, $P_i = (V_i, O_i, C_i)$ where $V_i \subseteq V$,
- S is a finite set of *states*,
- $\delta : S \times \Sigma \rightarrow 2^S$ is a *transition function*,
- $S_0 \subseteq S$ is a non-empty set of *initial states*,
- $F \subseteq S$ is a non-empty set of *final states*.

An MCA is a generalization of an ordinary finite-state automaton in which the alphabets are a finite set of patterns defined over a set of typed variables. Given a binding $\mathcal{B}$ for the variables $V = \{v_1, \ldots, v_k\}$, the finite-state automaton obtained by substituting $\mathcal{B}(v_i)$ for v_i for all $1 \leq i \leq k$ in $\mathcal{A}$ is denoted by $\mathcal{B}(\mathcal{A})$. Note that $\mathcal{B}(\mathcal{A})$ is a "vanilla" finite-state automaton. We explain this using an example. Consider the MCA $\mathcal{A}$ shown in Figure 10 with $V = \{A, B, C, D\}$. The automata obtained from $\mathcal{A}$ corresponding to the bindings $\mathcal{B}_1$ and $\mathcal{B}_2$ are shown in Figure 10. The uninterpreted variables in the MCA were introduced to handle obfuscation transformations based on register reassignment. The malicious code automaton corresponding to the code fragment shown in Figure 6 (from the Chernobyl virus) is depicted in Figure 11.




```
mov    esi, ecx
mov    eax, 0d601h
pop    edx
pop    ecx
```

$\mathcal{B}_1 = \{$ [A, esi], [B, ecx], [C, eax], [D, edx] $\}$

```
mov    esi, eax
mov    ebx, 0d601h
pop    ecx
pop    eax
```

$\mathcal{B}_2 = \{$ [A, esi], [B, eax], [C, ebx], [D, ecx] $\}$

Figure 10: Malicious code automaton for a Chernobyl virus code fragment, and instantiations with different register assignments, shown with their respective bindings.

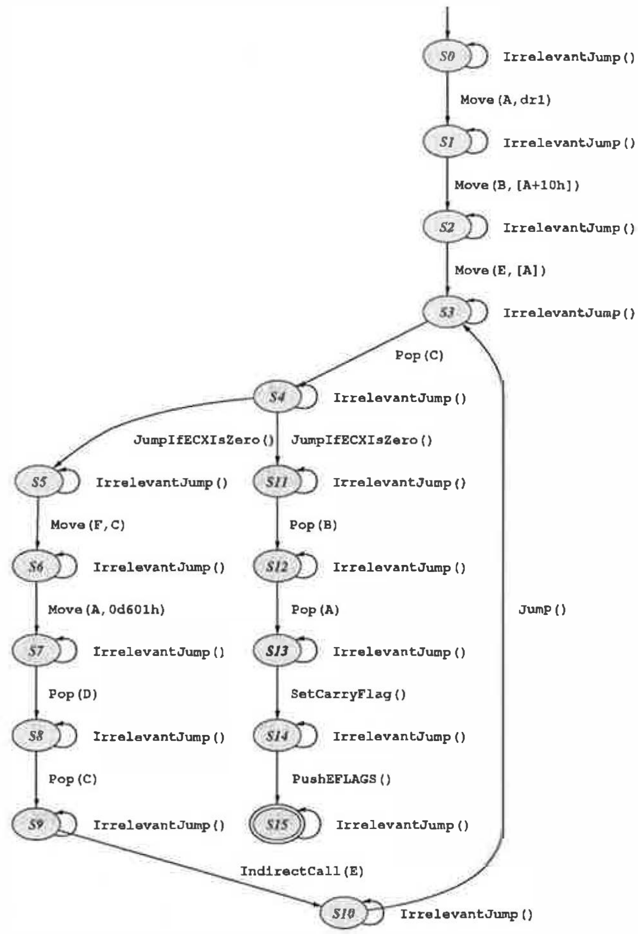


Figure 11: Malicious code automaton corresponding to code fragment from Figure 6.

7.2 Detector Operation

The detector takes as its inputs the annotated CFG P_Σ of a program procedure P and a malicious code automaton MCA $\mathcal{A} = (V, \Sigma, S, \delta, S_0, F)$. Note that the set of patterns Σ is used both to construct the annotated CFG and as the alphabet of the malicious code automaton. Intuitively, the detector determines whether there exists a malicious pattern that occurs in $\mathcal{A}$ and P_Σ. We formalize this intuitive notion. The annotated CFG P_Σ is a finite-state automaton where nodes are states, edges represent transitions, the node corresponding to the entry point is the initial state, and every node is a final state. Our detector determines whether the following language is empty:

$$L(P_\Sigma) \cap \left(\bigcup_{\mathcal{B} \in \mathcal{B}_{All}} L(\mathcal{B}(\mathcal{A})) \right)$$

In the expression given above, $L(P_\Sigma)$ is the language corresponding to the annotated CFG and $\mathcal{B}_{All}$ is the set of all bindings to the variables in the set V. In other words, the detector determines whether there exists a binding $\mathcal{B}$ such that the intersection of the languages P_Σ and $\mathcal{B}(\mathcal{A})$ is non-empty.

Our detection algorithm is very similar to the classic algorithm for determining whether the intersection of two regular languages is non-empty [22]. However, due to the presence of variables, we must perform unification during the algorithm. Our algorithm (Figure 12) combines the classic algorithm for computing the intersection of two regular languages with unification. We have implemented the algorithm as a data-flow analysis.

- For each node n of the annotated CFG P_A we associate pre and post lists L_n^{pre} and L_n^{post} respectively. Each element of a list is a pair $[s, \mathcal{B}]$, where s is the state of the MCA $\mathcal{A}$ and $\mathcal{B}$ is the binding of variables. Intuitively, if $[s, \mathcal{B}] \in L_n^{pre}$, then it is possible for $\mathcal{A}$ with the binding $\mathcal{B}$ (i.e. for $\mathcal{B}(\mathcal{A})$) to be in state s just before node n.
- **Initial condition:** Initially, both lists associated with all nodes except the start node n_0 are empty. The pre list

associated with the start node is the list of all pairs $[s, \emptyset]$, where s is an initial state of the MCA $\mathcal{A}$, and the post list associated with the start node is empty.

- **The do-until loop:** The do-until loop updates the pre and post lists of all the nodes. At the end of the loop, the worklist WS contains the set of nodes whose pre or post information has changed. The loop executes until the pre and post information associated with the nodes does not change, and a fixed point is reached. The join operation that computes L_i^{pre} takes the list of state-binding pairs from all of the L_j^{post} sets for program points preceding i and copies them to L_i^{pre} only if there are no repeated states. In case of repeated states, the conflicting pairs are merged into a single pair only if the bindings are compatible. If the bindings are incompatible, both pairs are thrown out.
- **Diagnostic feedback:** Suppose our algorithm returns a non-empty set, meaning a malicious pattern is common to the annotated CFG P_Σ and MCA $\mathcal{A}$. In this case, we return the sequence of instructions in the executable corresponding to the malicious pattern. This is achieved by keeping an additional structure with the algorithm. Every time the post list for a node n is updated by taking a transition in $\mathcal{A}$ (see the statement 14 in Figure 12), we store the predecessor of the added state, i.e., if $[\delta(s,\Gamma), \mathcal{B}_s \cup \mathcal{B}]$ is added to L_n^{post}, then we add an edge from s to $\delta(s,\Gamma)$ (along with the binding $\mathcal{B}_s \cup \mathcal{B}$) in the associated structure. Suppose we detect that L_n^{post} contains a state $[s, \mathcal{B}_s]$, where s is a final state of the MCA $\mathcal{A}$. Then we traceback the associated structure from s until we reach an initial state of $\mathcal{A}$ (storing the instructions occurring along the way).

8 Experimental Data

The three major goals of our experiments were to measure the execution time of our tool and find the false positive and negative rates. We constructed ten obfuscated versions of the four viruses. Let $V_{i,k}$ (for $1 \le i \le 4$ and $1 \le k \le 10$) denote the k-th version of the i-th virus. The obfuscated versions were created by varying the obfuscation parameters, e.g., number of nops and inserted jumps. For the i-th virus, $V_{i,1}$ denoted the "vanilla" or the unobfuscated version of the virus. Let M_1, M_2, M_3 and M_4 be the malicious code automata corresponding to the four viruses.

8.1 Testing Environment

The testing environment consisted of a Microsoft Windows 2000 machine. The hardware configuration included an AMD Athlon 1 GHz processor and 1 GB of RAM. We used CodeSurfer version 1.5 patchlevel 0 and IDA Pro version 4.1.7.600.

8.2 Testing on Malicious Code

We will describe the testing with respect to the first virus. The testing for the other viruses is analogous. First, we ran SAFE on the 10 versions of the first virus $V_{1,1}, \ldots, V_{1,10}$ with malicious code automaton M_1. This experiment gave us the false negative rate, i.e., the pattern corresponding to M_1 should be detected in all versions of the virus.

	Annotator		Detector	
	avg.	(std. dev.)	avg.	(std. dev.)
Chernobyl	1.444 s	(0.497 s)	0.535 s	(0.043 s)
z0mbie-6.b	4.600 s	(2.059 s)	1.149 s	(0.041 s)
f0sf0r0	4.900 s	(2.844 s)	0.923 s	(0.192 s)
Hare	9.142 s	(1.551 s)	1.604 s	(0.104 s)

Table 5: SAFE performance when checking obfuscated viruses for false negatives.

Next, we executed SAFE on the versions of the viruses $V_{i,k}$ with the malicious code automaton M_j (where $i \neq j$). This helped us find the false positive rate of SAFE.

In our experiments, we found that SAFE's false positive and negative rate were 0. We also measured the execution times for each run. Since IDA Pro and CodeSurfer were not implemented by us, we did not measure the execution times for these components. We report the average and standard deviation of the execution times in Tables 5 and 6.

	Annotator		Detector	
	avg.	(std. dev.)	avg.	(std. dev.)
z0mbie-6.b	3.400 s	(1.428 s)	1.400 s	(0.420 s)
f0sf0r0	4.900 s	(1.136 s)	0.840 s	(0.082 s)
Hare	1.000 s	(0.000 s)	0.220 s	(0.019 s)

Table 6: SAFE performance when checking obfuscated viruses for false positives against the Chernobyl/CIH virus.

8.3 Testing on Benign Code

We considered a suite of benign programs (see Section 8.3.1 for descriptions). For each benign program, we executed SAFE on the malicious code automaton corresponding to the four viruses. Our detector reported "negative" in each case, i.e., the false positive rate is 0. The average and variance of the execution times are reported in Table 7. As can be seen from the results, for certain cases the execution times are unacceptably large. We will address performance enhancements to SAFE in the future.

8.3.1 Descriptions of the Benign Executables

- tiffdither.exe is a command line utility in the *cygwin* toolkit version 1.3.70, a UNIX environment for Windows developed by Red Hat.

Input: A list of patterns $\Sigma = \{P_1, \ldots, P_r\}$, a malicious code automaton $\mathcal{A} = (V, \Sigma, S, \delta, S_0, F)$, and an annotated CFG $P_\Sigma =< N, E >$.
Output: $true$ if the program is likely infected, $false$ otherwise.
MALICIOUSCODECHECKING(Σ, $\mathcal{A}$, P_Σ)

$L_{n_0}^{pre} \leftarrow \{ [s, \emptyset] \mid s \in S_0 \}$, where $n_0 \in N$ is the entry node of P_Σ
foreach $n \in N \setminus \{n_0\}$ **do** $L_n^{pre} \leftarrow \emptyset$
foreach $n \in N$ **do** $L_n^{post} \leftarrow \emptyset$
$WS \leftarrow \emptyset$
do
  $WS_{old} \leftarrow WS$
  $WS \leftarrow \emptyset$
  foreach $n \in N$ // update pre information
    if $L_n^{pre} \neq \bigcup_{m \in Previous(n)} L_m^{post}$
      $L_n^{pre} \leftarrow \bigcup_{m \in Previous(n)} L_m^{post}$
      $WS \leftarrow WS \cup \{n\}$
  foreach $n \in N$ // update post information
    $NewL_n^{post} \leftarrow \emptyset$
    foreach $[s, \mathcal{B}_s] \in L_n^{pre}$
      foreach $[\Gamma, \mathcal{B}] \in Annotation(n)$ // follow a transition
        $\wedge\ Compatible(\mathcal{B}_s, \mathcal{B})$
        add $[\ \delta(s, \Gamma),\ \mathcal{B}_s \cup \mathcal{B}\ ]$ **to** $NewL_n^{post}$
    if $L_n^{post} \neq NewL_n^{post}$
      $L_n^{post} \leftarrow NewL_n^{post}$
      $WS \leftarrow WS \cup \{n\}$
until $WS = \emptyset$
return $\exists\, n \in N\ .\ \exists\, [s, \mathcal{B}_s] \in L_n^{post}\ .\ s \in F$

Figure 12: Algorithm to check a program model against a malicious code specification.

- winmine.exe is the Microsoft Windows 2000 Minesweeper game, version 5.0.2135.1.
- spyxx.exe is a Microsoft Visual Studio 6.0 Spy++ utility, that allows the querying of properties and monitoring of messages of Windows applications. The executable we tested was marked as version 6.0.8168.0.
- QuickTimePlayer.exe is part of the Apple QuickTime media player, version 5.0.2.15.

9 Conclusion and Future Work

We presented a unique view of malicious code detection as a obfuscation-deobfuscation game. We used this viewpoint to explore obfuscation attacks on commercial virus scanners, and found that three popular virus scanners were susceptible to these attacks. We presented a static analysis framework for detecting malicious code patterns in executables. Based upon our framework, we have implemented SAFE, a static analyzer for executables that detects malicious patterns in executables and is resilient to common obfuscation transformations.

For future work, we will investigate the use of theorem provers during the construction of the annotated CFG. For instance, SLAM [2] uses the theorem prover Simplify [16] for predicate abstraction of C programs. Our detection algorithm is context insensitive and does not track the calling context of the executable. We will investigate the use of the push-down systems, which would make our algorithm context sensitive. However, the existing PDS formalism does not allow uninterpreted variables, so it will have to be extended to be used in our context.

Availability

The SAFE prototype remains in development and we are not distributing it at this time. Please contact Mihai Christodorescu, `mihai@cs.wisc.edu`, for further updates.

Acknowledgments

We would like to thank Thomas Reps and Jonathon Giffin for providing us with invaluable comments on earlier drafts of the paper. We would also like to thank the members and collaborators of the Wisconsin Safety Analyzer (WiSA, `http://www.cs.wisc.edu/wisa`) research group for their insightful feedback and support throughout the development of this work.

References

[1] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *2002*

	Executable size	.text size	Procedure count	Annotator avg.	Annotator (std. dev.)	Detector avg.	Detector (std. dev.)
tiffdither.exe	9,216 B	6,656 B	29	6.333 s	(0.471 s)	1.030 s	(0.043 s)
winmine.exe	96,528 B	12,120 B	85	15.667 s	(1.700 s)	2.283 s	(0.131 s)
spyxx.exe	499,768 B	307,200 B	1,765	193.667 s	(11.557 s)	30.917 s	(6.625 s)
QuickTimePlayer.exe	1,043,968 B	499,712 B	4,767	799.333 s	(5.437 s)	160.580 s	(4.455 s)

Table 7: SAFE performance in seconds when checking clean programs against the Chernobyl/CIH virus.

IEEE Symposium on Security and Privacy (Oakland'02), pages 143–159, May 2002.

[2] T. Ball and S.K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN'01)*, volume 2057 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[3] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In *Advances in Cryptology (CRYPTO'01)*, volume 2139 of *Lecture Notes in Computer Science*, pages 1 – 18. Springer-Verlag, August 2001.

[4] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2), 1996.

[5] CERT Coordination Center. Denial of service attacks, 2001. `http://www.cert.org/tech_tips/denial_of_service.html` (Last accessed: 3 February 2003).

[6] S. Chandra and T.W. Reps. Physical type checking for C. In *ACM SIGPLAN - SIGSOFT Workshop on Program Analysis For Software Tools and Engineering (PASTE'99)*, pages 66 – 75. ACM Press, September 1999.

[7] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *9th ACM Conference on Computer and Communications Security (CCS'02)*. ACM Press, November 2002.

[8] B.V. Chess. Improving computer security using extending static checking. In *2002 IEEE Symposium on Security and Privacy (Oakland'02)*, pages 160–173, May 2002.

[9] D.M. Chess and S.R. White. An undetectable computer virus. In *Proceedings of Virus Bulletin Conference*, 2000.

[10] F. Cohen. Computer viruses: Theory and experiments. *Computers and Security*, 6:22 – 35, 1987.

[11] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Sciences, The University of Auckland, July 1997.

[12] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98)*. ACM Press, January 1998.

[13] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE'00)*, pages 439–448. ACM Press, 2000.

[14] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM Symposium on Principles of Programming Languages (POPL'78)*, pages 84 – 96. ACM Press, January 1978.

[15] D. W. Currie, A. J. Hu, and S. Rajan. Automatic formal verification of dsp software. In *Proceedings of the 37th ACM IEEE Conference on Design Automation (DAC'00)*, pages 130–135. ACM Press, 2000.

[16] D. Detlefs, G. Nelson, and J. Saxe. The simplify theorem prover. `http://research.compaq.com/SRC/esc/simplify.html` .

[17] U. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *2000 IEEE Symposium on Security and Privacy (Oakland'00)*, pages 246–255, May 2000.

[18] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Proceedings of the 12th International Conference on Computer-Aided Verification (CAV'00)*, volume 1855 of *Lecture Notes in Computer Science*, pages 232–247. Springer-Verlag, July 2000.

[19] X. Feng and Alan J. Hu. Automatic formal verification for scheduled VLIW code. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems - Software and Compilers for Embedded Systems (LCTES/SCOPES'02)*, pages 85–92. ACM Press, 2002.

[20] M. Fitting. *First-Order Logic and Automated Theorem Proving*. Springer-Verlag, 1996.

[21] J. T. Giffin, S. Jha, and B. P. Miller. Detecting manipulated remote call streams. In *Proceedings of the 11th USENIX Security Symposium (Security'02)*. USENIX Association, August 2002.

[22] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2001.

[23] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, January 1990.

[24] GrammaTech Inc. Codesurfer – code analysis and understanding tool. `http://www.grammatech.com/products/codesurfer/index.html` (Last accessed: 3 February 2003).

[25] T. Jensen, D.L. Metayer, and T. Thorn. Verification of control flow based security properties. In *1999 IEEE*

Symposium on Security and Privacy (Oakland'99), May 1999.

[26] E. Kaspersky. *Virus List Encyclopaedia*, chapter Ways of Infection: Viruses without an Entry Point. Kaspersky Labs, 2002. `http://www.viruslist.com/eng/viruslistbooks.asp?id=32&key=0000100007000020000100003` (Last accessed: 3 February 2003).

[27] Kaspersky Labs. `http://www.kasperskylabs.com` (Last accessed: 3 February 2003).

[28] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323 – 337, December 1992.

[29] R.W. Lo, K.N. Levitt, and R.A. Olsson. MCF: A malicious code filter. *Computers & Society*, 14(6):541–566, 1995.

[30] G. McGraw and G. Morrisett. Attacking malicious code: Report to the Infosec research council. *IEEE Software*, 17(5):33 – 41, September/October 2000.

[31] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. The spread of the Sapphire/Slammer worm. `http://www.caida.org/outreach/papers/2003/sapphire/sapphire.html` (Last accessed: 3 February 2003).

[32] G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based Typed Assembly Language. In Xavier Leroy and Atsushi Ohori, editors, *1998 Workshop on Types in Compilation*, volume 1473 of *Lecture Notes in Computer Science*, pages 28 – 52. Springer-Verlag, March 1998.

[33] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98)*, pages 85 – 97. ACM Press, January 1998.

[34] S.S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[35] E.M. Myers. A precise interprocedural data flow algorithm. In *Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages (POPL'81)*, pages 219 – 230. ACM Press, January 1981.

[36] C. Nachenberg. Polymorphic virus detection module. *United States Patent # 5,696,822*, December 9, 1997.

[37] C. Nachenberg. Polymorphic virus detection module. *United States Patent # 5,826,013*, October 20, 1998.

[38] G. C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*, pages 83–94. ACM Press, June 2000.

[39] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In *Proceedings of the 8th International Conference on Computer-Aided Verification (CAV'96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer-Verlag, August 1996.

[40] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*, pages 49–61. ACM Press, January 1995.

[41] M. Samamura. *Expanded Threat List and Virus Encyclopaedia*, chapter W95.CIH. Symantec Antivirus Research Center, 1998. `http://securityresponse.symantec.com/avcenter/venc/data/cih.html` (Last accessed: 3 February 2003).

[42] DataRescue sa/nv. IDA Pro – interactive disassembler. `http://www.datarescue.com/idabase/` (Last accessed: 3 February 2003).

[43] S. Staniford, V. Paxson, and N. Weaver. How to 0wn the internet in your spare time. In *Proceedings of the 11th USENIX Security Symposium (Security'02)*, pages 149 – 167. USENIX, USENIX Association, August 2002.

[44] P. Ször and P. Ferrie. Hunting for metamorphic. In *Proceedings of Virus Bulletin Conference*, pages 123 – 144, September 2001.

[45] TESO. burneye elf encryption program. `https://teso.scene.at` (Last accessed: 3 February 2003).

[46] D. Wagner and D. Dean. Intrusion detection via static analysis. In *2001 IEEE Symposium on Security and Privacy (Oakland'01)*, May 2001.

[47] R. Wang. Flash in the pan? *Virus Bulletin*, July 1998. Virus Analysis Library.

[48] Z. Xu. *Safety-Checking of Machine Code*. PhD thesis, University of Wisconsin, Madison, 2000.

[49] z0mbie. Automated reverse engineering: Mistfall engine. `http://z0mbie.host.sk/autorev.txt` (Last accessed: 3 February 2003).

[50] z0mbie. RPME mutation engine. `http://z0mbie.host.sk/rpme.zip` (Last accessed: 3 February 2003).

[51] z0mbie. z0mbie's homepage. `http://z0mbie.host.sk` (Last accessed: 3 February 2003).

Notes

[1]Note that the subroutine address computation had to be updated to take into account the new nops. This is a trivial computation and can be implemented by adding the number of inserted nops to the initial offset hard-coded in the virus-morphing code.

[2]Most executable formats require that the various sections of the executable file start at certain aligned addresses, to respect the target platform's idiosyncrasies. The extra space between the end of one section and the beginning of the next is usually padded with nulls.

[3]We use one-way matching which is simpler than full unification. Note that the instruction sequence does not contain any variables. We instantiate variables in the pattern so that they match the corresponding terms in the instruction sequence.

SSL splitting: securely serving data from untrusted caches

Chris Lesniewski-Laas and M. Frans Kaashoek
{ctl,kaashoek}@mit.edu
Laboratory for Computer Science
Massachusetts Institute of Technology

Abstract

A popular technique for reducing the bandwidth load on Web servers is to serve the content from proxies. Typically these hosts are trusted by the clients and server not to modify the data that they proxy. SSL splitting is a new technique for guaranteeing the integrity of data served from proxies without requiring changes to Web clients. Instead of relaying an insecure HTTP connection, an SSL splitting proxy simulates a normal Secure Sockets Layer (SSL) [7] connection with the client by merging authentication records from the server with data records from a cache. This technique reduces the bandwidth load on the server, while allowing an unmodified Web browser to verify that the data served from proxies is endorsed by the originating server.

SSL splitting is implemented as a patch to the industry-standard OpenSSL library, with which the server is linked. In experiments replaying two-hour `access.log` traces taken from LCS Web sites over an ADSL link, SSL splitting reduces bandwidth consumption of the server by between 25% and 90% depending on the warmth of the cache and the redundancy of the trace. Uncached requests forwarded through the proxy exhibit latencies within approximately 5% of those of an unmodified SSL server.

1 Introduction

Caching Web proxies are a proven technique for reducing the load on centralized servers. For example, an Internet user with a Web site behind an inexpensive DSL line might ask a number of well-connected volunteers to proxy the content of the Web site to provide higher throughput. In today's practice, these proxies must be trusted by both the client and the server to return the data to the client's queries, unmodified.

Previous content delivery systems that guarantee the integrity of the data served by proxies require changes to the client software (e.g., to support SFSRO [8]), or use application-specific solutions (e.g., RPM with PGP signatures [25]). The former have not seen wide application due to lack of any existing client base. The latter are problematic due to PKI bootstrapping issues and due to the large amount of manual intervention required for their proper use.

Our goal is to guarantee the integrity of data served by the proxy without requiring changes to clients. Our approach is to exploit the existing, widely-deployed browser support for the Secure Sockets Layer (SSL) protocol [7]. We modify the server end of the SSL connection by splitting it (see Figure 1): the central server sends the SSL record authenticators, and the proxy merges them with a stream of message payloads retrieved from the proxy's cache. The merged data stream that the proxy sends to the client is indistinguishable from a normal SSL connection between the client and the server. We call this technique of splitting the authenticator and data records *SSL splitting*.

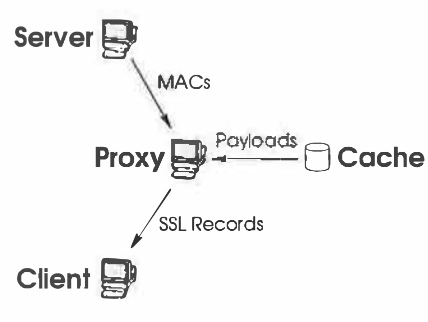


Figure 1: Data flow in SSL splitting.

SSL splitting cleanly separates the roles of the server and the proxy: the server, as "author", originates and signs the correct data, and the proxy, as "distribution channel",

This research was partially supported by MIT Project Oxygen and the IRIS project (`http://project-iris.net/`), funded by the National Science Foundation under Cooperative Agreement No. ANI-0225660.

serves the data to clients. SSL splitting cannot provide confidentiality, since the proxy must have access to the encryption keys shared between client and server to re-encrypt the merged stream; thus, our technique is only useful for distributing public data. SSL splitting also doesn't reduce the CPU load on the server, since the server is still involved in establishing the SSL connection, which requires a public-key operation on the server. The primary advantage of SSL splitting is that it reduces the bandwidth load on the server.

Our primary application for SSL splitting is *Barnraising*, a cooperative Web cache. We anticipate that cooperative content delivery will be useful to bandwidth-hungry Web sites with limited central resources, such as software distribution archives and media artists' home pages. Currently, such sites must be mirrored by people known and trusted by the authors, since a malicious mirror can sabotage the content; Barnraising would allow such sites to harness the resources of arbitrary hosts on the Internet, while still guaranteeing the integrity of the data. Barnraising could also be used by *ad hoc* cooperatives of small, independent Web sites, to distribute the impact of localized load spikes.

The contributions of this paper are: the design of the SSL splitting technique, including the simple protocol between server and proxy to support SSL splitting; an implementation of SSL splitting based on the freely available OpenSSL library; a new, grassroots content-distribution system, *Barnraising*, which applies SSL splitting to distribute bandwidth load; and experiments that show that SSL splitting has CPU costs similar to SSL, but saves server bandwidth, and improves download times for large files.

2 Goals

Our main goal is to guarantee that public data served by caching Web proxies is endorsed by the originating server. Anybody with an inexpensive DSL line should be able to author content and distribute it from his own Web server. To allow this limited connection to support a higher throughput, authors can leverage the resources of well-connected volunteers acting as mirrors of the site's content. However, since the authors may not fully trust the volunteers, we must provide an end-to-end authenticity and freshness guarantee: the content accepted by a client must be the latest version of the content published by the author.

Our second goal is to provide the data integrity with minimal changes to the existing infrastructure. More specifically, our goal is a solution that does not require any client-side changes and minimal changes to a server. To satisfy this goal, we exploit the existing support for SSL, by splitting the server end of the connection.

Confidentiality is not a goal. The intended use is to distribute public, popular data from bandwidth-limited servers. Most of the content on the Web is not secret, and moreover, most pages that require secrecy are dynamically generated and hence not cacheable. Lack of perfect confidentiality is also an inevitable consequence of caching, because any caching proxy must be able to tell when two clients have downloaded the same file; this requirement violates ciphertext indistinguishability.

SSL splitting does not provide all the benefits of traditional, insecure mirroring. While it improves the bandwidth utilization of the central site, it incurs a CPU load similar to a normal SSL server. In addition, it does not improve the redundancy of the site, since the central server must be available in order to authenticate data. Redundant central servers must be employed to ensure continued service in the face of server failure or network partition.

3 Design of SSL splitting

The key idea behind SSL splitting is that a stream of SSL records is separable into a data component and an authenticator component. As long as the record stream presented to the client has the correct format, the two components can arrive at the proxy by different means. In particular, a proxy can cache data components, avoiding the need for the server to send the data in full for every client.

While SSL splitting does not require changes to the client software, it does require a specialized proxy and modifications to the server software. The modified server and proxy communicate using a new protocol that encapsulates the regular SSL protocol message types and adds two message types of its own.

3.1 SSL overview

The Secure Sockets Layer protocol provides end-to-end mutual authentication and confidentiality at the transport layer of stream-based protocols [7]. A typical SSL connection begins with a handshake phase, in which the server authenticates itself to the client and shared keys are generated for the connection's symmetric ciphers. The symmetric keys generated for authentication are distinct from those generated for confidentiality, and the keys generated for the server-to-client data stream are

distinct from those generated from the client-to-server stream.

After completing the handshake, the server and client exchange data asynchronously in both directions along the connection. The data is split into records of 2^{14} bytes or less. For each record, the sender computes a Message Authentication Code (MAC) using the symmetric authentication keys; this enables the receiver to detect any modification of the data in transit. SSL can provide confidentiality as well as integrity: records may be encrypted using the shared symmetric encryption keys. Although SSL generates the keys for both directions from the same "master secret" during the handshake phase, the two directions are subsequently independent: the client or server's outgoing cipher state depends only on the previous records transmitted by that party.

3.2 Interposing a proxy

To access a site using SSL splitting, a Web browser must connect to a proxy using HTTP over SSL/TLS (HTTPS [18]). The server may have redirected the client to the proxy via any mechanism of its choice, or the proxy may have already been on the path between the browser and the server.

The proxy relays the client's connection setup messages to the server, which in turn authenticates itself to the client via the proxy. Once the SSL connection is set up, the server starts sending application data: for each record, it sends the message authentication code (MAC) along with a short unique identifier for the payload. (See Figure 2.) Using the identifier, the proxy looks up the payload in its local cache, splices this payload into the record in place of the identifier, and relays this reconstructed record to the client. The client verifies the integrity of the received record stream, which is indistinguishable from the stream that would have been sent by a normal SSL server.

Since SSL is resistant to man-in-the-middle attacks, and the proxy is merely a man-in-the-middle with respect to the SSL handshake, the SSL authentication keys are secret from the proxy. Only the server and the client know these keys, which enable them to generate and verify the authentication codes protecting the connection's end-to-end integrity and freshness.

3.3 Proxy-server protocol extensions

When a client initiates an HTTPS connection to an SSL-splitting proxy, the proxy immediately connects to the server using the specialized proxy-server protocol. This protocol defines three message types. The first type of message, *verbatim*, is a regular SSL record, passed transparently through the proxy from the client to the server, or vice versa. The second type of message, *stub*, is a compact representation of an SSL record: it contains a MAC authenticator and a short unique identifier for the payload. The third type of message, *key-expose*, communicates an encryption key from the server to the proxy.

When the proxy receives SSL records from the client, it forwards them directly to the server using the *verbatim* message. The server, however, may choose to compress the data it sends by using the *stub* message format. When the proxy receives such a message from the server, it looks up the data block identified by the message in its local cache. It then reconstructs a normal SSL record by splicing the MAC authenticator from the stub record together with the payload from the cache, and forwards the resulting valid SSL record to the client.

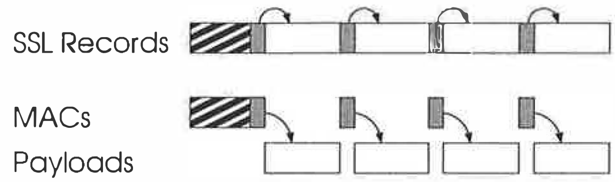


Figure 2: Decomposition of an SSL stream into authenticators and payloads. The striped box represents the SSL handshake, which is handled by the server. The shaded boxes represent authenticators, while the white boxes represent payloads.

3.4 Dropping the encryption layer

A proxy can properly forward *stub* messages only if it is able to encode the resulting normal SSL records. If the client and server use SSL with its end-to-end encryption layer enabled, however, the proxy cannot send validly encrypted messages. End-to-end encryption inherently foils caching, because a proxy will not be able to determine when the same data is downloaded by different clients. Therefore, to achieve bandwidth compression, confidentiality—with respect to the proxy—must be abandoned.

The correct way to eliminate SSL's encryption layer is to negotiate, during the handshake phase, an authentication-only cipher suite such as `SSL_RSA_WITH_NULL_SHA`; this is usually done with a Web server configuration setting. When such a cipher

suite is in use, no confidentiality is provided for SSL records sent in either direction; only data authentication is provided.

Unfortunately, this straightforward approach does not achieve full compatibility with the existing installed client base, because current versions of many popular Web browsers, such as Netscape and Internet Explorer, ship with authentication-only cipher suites disabled. The SSL splitting protocol provides a work-around for this problem, which can be enabled by the server administrator. If the option is enabled and a client does not offer an authentication-only cipher suite, the server simply negotiates a normal cipher suite with the client, and then intentionally exposes the server-to-client encryption key and initialization vector (IV) to the proxy using the *key-expose* message.

SSL computes the encryption key, IV, and MAC key as independent pseudo-random functions (PRF) of the master secret. Because SSL uses different PRFs for each key, revealing the cipher key and IV to the proxy does not endanger the MAC key or any of the client-to-server keys [19, p. 165]. Hence, *key-expose* preserves the data-authentication property.

When the *key-expose* feature is turned on and an encrypted cipher suite is negotiated, the client-to-server encryption keys are withheld from the proxy; thus, it would be possible to develop an application in which the information sent by a client was encrypted, while the content returned by the server was unencrypted and cacheable by the proxy. However, unless the application is carefully designed, there is a danger of leaking sensitive data in the server's output, and so we do not recommend the use of SSL splitting in this mode without very careful consideration of the risks.

If necessary, an application can restrict the set of hosts with access to the cleartext to the client, the server, and the proxy, by encrypting the proxy-server connection. For example, one could tunnel the proxy-server connection through a (normal) SSL connection. Of course, this feature would be useful only in applications where the proxy can be trusted not to leak the cleartext, intentionally or not; as above, we do not recommend this mode of operation.

3.5 Server-proxy signaling

SSL splitting does not mandate any particular cache coherency mechanism, but it does affect the factors that make one mechanism better than another. In particular, caching with SSL splitting is at the SSL record level rather than at the file level. In theory, a single file could be split up into records in many different ways, and this would be a problem for the caching mechanism; however, in practice, a particular SSL implementation will always split up a given data stream in the same way.

Deciding which records to encode as *verbatim* records and which to encode as *stub* records can be done in two ways. The server can remember which records are cached on each proxy, and consult an internal table when deciding whether to send a record as a stub. However, this has several disadvantages. It places a heavy burden on the server, and does not scale well to large numbers of proxies. It does not give the proxies any latitude in deciding which records to cache and which to drop, and proxies must notify the server of any changes in their cached set. Finally, this method does not give a clear way for the proxies to initialize their cache.

Our design for SSL splitting uses a simpler and more robust method. The server does not maintain any state with respect to the proxies; it encodes records as *verbatim* or *stub* without regard to the proxy. If the proxy receives a *stub* that is not in its local cache, it triggers a cache miss handler, which uses a simple, HTTP-like protocol to download the body of the record from the server. Thus, the proxies are self-managing and may define their own cache replacement policies. This design requires the server to maintain a local cache of recently-sent records, so that it will be able to serve cache miss requests from proxies. Although a mechanism similar to TCP acknowledgements could be used to limit the size of the server's record cache, a simple approach that suffices for most applications is to pin records in the cache until the associated connection is terminated.

The cache-miss design permits the server to use an arbitrary policy to decide which records to encode as *stub* and hence to make available for caching; the most efficient policy depends on the application. For HTTPS requests, an effective policy is to cache all application-data records that do not contain HTTP headers, since the headers are dynamic and hence not cacheable. Most of the records in the SSL handshake contain dynamic elements and hence are not cacheable; however, the server's certificates are cacheable. Caching the server certificates has an effect similar to that of the "fast-track" optimization described in [22], and results in an improvement in SSL handshake performance. This caching is especially beneficial for requests of small files, where the latency is dominated by the SSL connection time.

Since the *stub* identifier is unique and is not reused, there is no need for a mechanism to invalidate data in the

cache. When the file referenced by an URL changes, the server sends a different stream of identifiers to the proxy, which does not know anything about the URL at all. In contrast, normal caching Web proxies [24] rely on invalidation timeouts for a weak form of consistency.

4 Implementation

Our implementation of SSL splitting consists of a self-contained proxy module and a patched version of the OpenSSL library, which supports SSL version 3 [7] and Transport Layer Security (TLS) [4].

4.1 Server: modified OpenSSL

We chose to patch OpenSSL, rather than developing our own protocol implementation, to simplify deployment. Any server that uses OpenSSL, such as the popular Web server Apache, works seamlessly with the SSL splitting protocol. In addition, the generic SSLizing proxy STunnel, linked with our version of OpenSSL, works as an SSL splitting server: using this, one can set up SSL splitting for servers that don't natively understand SSL. This allows SSL splitting to be layered as an additional access method on top of an existing network resource.

Our modified version of OpenSSL intercepts the record-encoding routine `do_ssl_write` and analyzes outgoing SSL records to identify those that would benefit from caching. Our current implementation tags non-header application-data records and server certificate records, as described in Section 3.5.

Records that are tagged for caching are hashed to produce a short *digest* payload ID, and the bodies of these records are *published* to make them available to proxies that do not have them cached. While publishing may take many application-specific forms, our implementation simply writes these payloads into a cache directory on the server; a separate daemon process serves this directory to proxies.

Records that are not tagged for caching use the *literal* payload "ID" encoding. Whether or not the record is tagged for caching, the library encodes the record as a *stub* message; this design choice simplifies the code.

Because the server may have to ship its encryption key and IV to the proxy, the modified OpenSSL library contains additional states in the connection state machine to mediate the sending of the *key-expose* message. The server sends this message immediately after any `change_cipher_spec` record, since at this point the connection adopts a new set of keys.

4.2 Proxy

The proxy is simple: it forks off two processes to forward every accepted connection. It is primarily written in OO Perl5, with the performance-critical block cipher and CBC mode implementation in C. The proxy includes a pluggable cache hierarchy: when a cache lookup for a payload ID fails, it can poll outside sources, such as other proxies, for the missing data. If all else fails, the server itself serves as an authority of last resort. Only if none of these have the payload will the proxy fail the connection.

The proxy could also replace *verbatim* messages from the client to the server with *stub* messages to avoid sending the complete request to the server. Because the client's data consists primarily of HTTP GET requests, however, which are already short and are not typically repeated, our current proxy doesn't do so. In the future, though, we may explore a proxy that compresses HTTP headers using *stub* messages.

4.3 Message formats

Figures 3 and 4 show the format of *verbatim* and *stub* message in the same notation as the SSL specification. The main difference between *verbatim* and *stub* is that in *stub* the payload is split into a compact encoding of the data and a MAC authenticator for the data. Also, a stub record is never encrypted, since the proxy would have to decrypt it anyway in order to manipulate its contents.

We have defined two types of encoding for the payload. The *literal* encoding is the identity function; this encoding is useful for software design reasons, but is functionally equivalent to a *verbatim* SSL record.

The *digest* encoding is a SHA-1 [5] digest of the payload contents. This encoding provides effectively a unique identifier that depends only on the payload. This choice is convenient for the server, and allows the proxy to store payloads from multiple independent servers in a single cache without concern about namespace collisions.

There are many alternative ID encodings possible with the given *stub* message format; for example, a simple serial number would suffice. The serial number, however, has only small advantages over a message digest. A serial number is guaranteed to be unique, unlike a digest. On the other hand, generating serial numbers requires servers to maintain additional state, and places an onus upon proxies to separate the caches corresponding to multiple servers; both of these result in greater complexity than the *digest* encoding.

```
enum { ccs(0x14), alert(0x15), handshake(0x16), data(0x17) } ContentType;
struct {
 ContentType     content_type;           one byte long
 uint8           ssl_version[2];
 opaque          encrypted_data_and_mac<0..2^14+2048>;
                                         includes implicit 2-byte length field
} VerbatimMessage;
```

Figure 3: Format of a verbatim SSL record.

```
enum { s_ccs(0x94), s_alert(0x95), s_handshake(0x96), s_data(0x97) } StubContentType;
enum { literal(1), digest(2), (2^16-1) } IDEncoding;
struct {
 StubContentType content_type;           = verbatim.content_type | 0x80
 uint8           ssl_version[2];
 uint16          length;                 = 6 + length(id) + length(mac)
 IDEncoding      encoding;
 opaque          id<0..2^14+2048>;
 opaque          mac<0..2^14+2048>;
} StubMessage;
```

Figure 4: Format of a stub message.

Another alternative is to use as the encoding a compressed representation of the payload. While this choice would result in significant savings for text-intensive sites, it would not benefit image, sound, or video files at all, since most media formats are already highly compressed. For this reason, we have not implemented this feature.

The *key-expose* message is used to transmit the server encryption key and IV to the proxy (see Figure 5), if this feature is enabled. In our implementation, *stub* records are sent in the clear to the proxy, which encrypts them before sending them to the client.

5 Cooperative Web caching using SSL splitting

Using SSL splitting, we have developed *Barnraising*, a cooperative Web caching system consisting of a dynamic set of "volunteer" hosts. The purpose of this system is to improve the throughput of bandwidth-limited Web servers by harnessing the resources of geographically diverse proxies, without trusting those proxies to ensure that the correct data is served.

5.1 Joining and leaving the proxy set

Volunteers join or leave the proxy set of a site by contacting a *broker* server, which maintains the volunteer database and handles redirecting client requests to volunteers.

Volunteer hosts do not locally store any configuration information, such as the SSL splitting server's address and port number. These parameters are supplied by the broker in response to the volunteer's *join* request, which simply specifies an identifying URI of the form `barnraising://broker.domain.org/some/site/name`.

This design enables a single broker to serve any number of Barnraising-enabled Web sites, and permits users to volunteer for a particular site given only a short URI for that site. Since configuration parameters are under the control of the broker, they can be changed without manually reconfiguring all proxies, allowing sites to upgrade transparently.

The broker represents a potential bottleneck for the system, and it could be swamped by a large number of simultaneous join or leave requests. However, since the join/leave protocol is lightweight, this is unlikely to be a performance issue under normal operating conditions. If the load incurred by requests is high compared to the actual SSL splitting traffic, the broker can simply rate-limit them until the proxy set stabilizes at a smaller size.

5.2 Redirection

Barnraising currently employs the DNS redirection method [11], but could be modified to support other

```
enum { key_expose(0x58) } KeyExposeContentType;
struct {
 KeyExposeContentType content_type;
 uint8                ssl_version[2];
 uint16               length;          = 4 + length(key) + length(iv)
 opaque               key<0..2^14+2048>;
 opaque               iv<0..2^14+2048>;
} KeyExposeMessage;
```

Figure 5: Format of key-expose message.

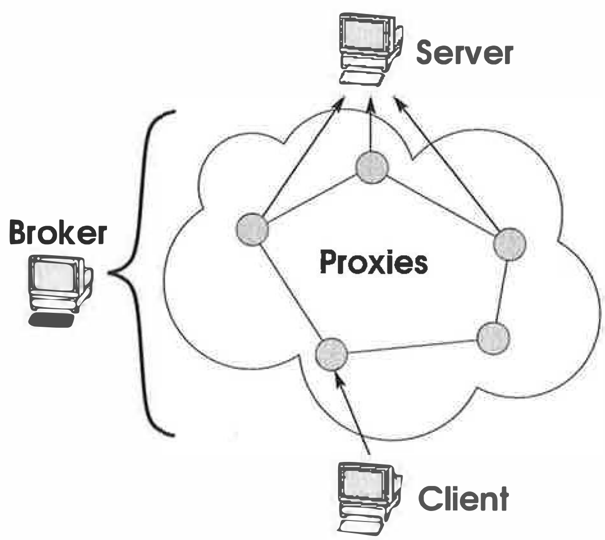


Figure 6: Proxy set for a site using Barnraising.

techniques, such as URL rewriting [10, 1]. Barnraising's broker controls a `mysql` [16] database, from which a `mydns` [15] server processes DNS requests.

Consider a client resolving an URL `https://www.domain.org/foo/`. If `www.domain.org` is using Barnraising, the DNS server for `domain.org` is controlled by the broker, which will resolve the name to the IP address of a volunteer proxy.

We chose redirection using DNS because it maintains HTTPS reference integrity — that is, it guarantees that hyperlinks in HTML Web pages dereference to the intended destination pages. HTTPS compares the hostname specified in the `https` URL with the certificate presented by the server. Therefore, when a client contacts a proxy via a DNS name, the certificate presented to the client, by the server, via the proxy, must match the domain name. When DNS redirection is used, the domain name will be of the form `www.domain.org`, and will match the domain name in the certificate.

5.3 Distributing the Web cache

The client will initiate an HTTPS connection to the proxy, which will forward that request, using SSL splitting, to the server. Since frequently-accessed data will be served out of the proxy's cache, the central server's bandwidth usage will be essentially limited to the SSL handshake, MAC stream, and payload IDs.

To increase the set of cached payloads available to a proxy, while decreasing the local storage requirements, proxies could share the cache among themselves. In this design, volunteer nodes would join a wide-area Distributed Hash Table (DHT) [3] comprising all of the volunteers for a given Web site. When lookups in the local cache fail, nodes could attempt to find another volunteer with the desired data item by looking for the data in the DHT. If that fails too, the proxy would contact the central server.

Blocks in the DHT are named by the same cryptographic hash used for *stub* IDs. This decision allows correctly-operating volunteers to detect and discard any invalid blocks that a malicious volunteer might have inserted in the DHT.

5.4 Deploying Barnraising

Barnraising is designed to be initially deployed as a transparent layer over an existing Web site, and incrementally brought into the core of the Web server. Using STunnel linked with our patched OpenSSL library, an SSL splitting server that proxies an existing HTTP server can be set up on the same or a different host; this choice allows Barnraising to be tested without disrupting existing services. If the administrator later decides to move the SSL splitting server into the core, he can use Apache linked with SSL-splitting OpenSSL.

The broker requires a working installation of `mysql` and `mydns`; since it has more dependencies than the server, administrators may prefer to use an existing third-party

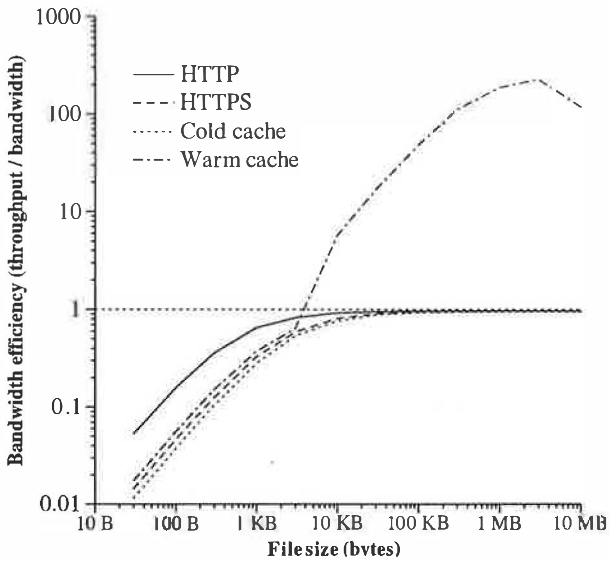


Figure 7: Ratio of throughput achieved to bandwidth used, when retrieving a single file from the server.

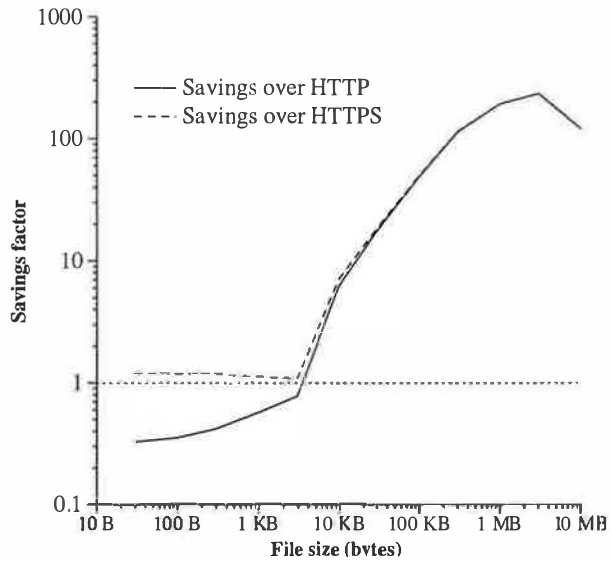


Figure 8: Bandwidth savings of SSL splitting over HTTP and HTTPS.

broker while testing Barnraising. This choice brings the third party into the central trust domain of the server; much like a traditional mirror, it is a role which can only be filled by a reputable entity.

The utility of Barnraising will be limited by the volunteer proxies that join it. Therefore, the proxy software has been designed to be simple to install and use, requiring only a single URI to configure. Since volunteers will be able to download from each other, they will have better performance than regular clients, giving users an incentive to install the software. In the long term, we hope to incorporate more efficient authentication schemes, such as SFSRO [8], into the volunteer code, using the installed base of SSL splitting software to bootstrap the more technically sophisticated systems.

6 Evaluation

This section presents microbenchmarks and trace-based experiments to test the effectiveness and practicality of SSL splitting. The results of these experiments demonstrate that SSL splitting decreases the bandwidth load on the server, and that the performance with respect to uncached files is similar to vanilla SSL.

6.1 General experimental setup

For the experiments we used the Apache web server (version 1.3.23), linked with mod_ssl (version 2.8.7), and OpenSSL (version 0.9.6), running under Linux 2.4.18 on a 500 MHz AMD K6. This server's network connection is a residential ADSL line with a maximum upstream bandwidth of 160 kbps. The client was a custom asynchronous HTTP/HTTPS load generator written in C using OpenSSL, running under FreeBSD 4.5 on a 1.2 GHz Athlon. The proxy, when used, ran under FreeBSD 4.5 on a 700 MHz Pentium III. Both the client and the proxy were on a 100 Mbps LAN, with a 100 Mbps uplink.

6.2 Bandwidth savings for a single file

Since SSL splitting caches files at the record level, and a cached record costs a fixed amount to transmit, we would expect the compression level to depend on the file size. A series of short microbenchmarks consisting of a single file download confirms that SSL splitting is far more effective at caching large files than small files. Figure 7 shows the bandwidth efficiency, calculated as the ratio of file throughput to bandwidth consumption, of HTTP, HTTPS, uncached SSL splitting, and cached SSL splitting. (The dotted line at 1 represents the theoretical performance of an ideal non-caching protocol with zero overhead.)

Figure 8 shows this data as the "savings factor", the ratio of bandwidth consumed by SSL splitting (with a warm cache) to that consumed by HTTP or HTTPS to transmit the same file. For 100-byte files, plain HTTP has about one-third the cost of SSL splitting, since the bandwidth

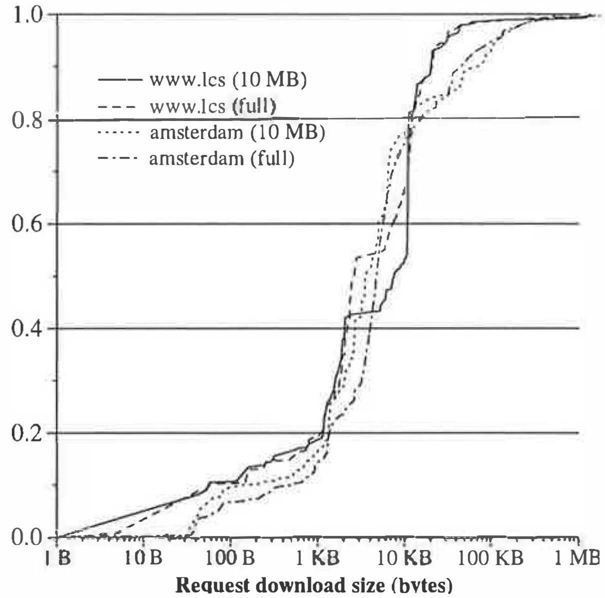


Figure 9: CDF of request sizes in www.lcs and amsterdam traces.

cost for small files is dominated by connection setup; on the other hand, for one-megabyte files, SSL splitting has a bandwidth savings of 99.5% over HTTP.

There are two artifacts evident in these graphs. They show a sharp curve upward at the 3,000 byte point; the reason for this artifact is that Apache sends all of the HTTP headers and file data in a single SSL record for files smaller than roughly 4,000 bytes, but sends the HTTP headers in a separate record from the file data for larger files. There is also a sharp performance drop between 3 megabytes and 10 megabytes: this is because, for files larger than 2^{22} bytes, Apache sends the file data in records of 8,192 bytes, instead of the maximum record size of 16,384 bytes.

6.3 Bandwidth savings for trace-driven loads

The next set of experiments uses traces from two Web servers to evaluate SSL splitting on realistic workloads.

6.3.1 Web trace files

The Web traces were derived from several-month `access.log` files taken from two departmental Web servers, `www.lcs.mit.edu` and `amsterdam.lcs.mit.edu`. To convert the access logs into replayable traces, all non-GET, non-status-200 queries were filtered out, URLs were canonicalized, and every (*URL*, *size*) pair was encoded as a unique URL. The server tree was then populated with files containing random bytes.

The `www.lcs` trace, which is seven months long, contains 109 GB of downloads from a server with 10.6 GB of files; the `amsterdam` trace, which is nine months long, contains 270 GB of downloads from 77 GB of files. Analyzing randomly-chosen chunks of various lengths from `www.lcs` showed that most repetition is long-term: a day's trace (typically about 100 MB of data transfer) has a repetition factor which varies between 1.5 and 3, while a month is compressible by about a factor of 4, and the whole trace is compressible by a factor of 10. This data suggests that having proxies keep blocks around for a long time will pay off, and supports a design in which proxies are organized into a DHT, since this allows them to store more unique blocks for a longer period of time.

To keep running experiments manageable over an ADSL line, we selected a typical daytime chunk representing approximately 10 MB of transfers from each trace. Figure 9 shows the distribution of request sizes in each trace chunk, along with the distribution in the full traces. The `www.lcs` chunk represents 4.43 MB of files and 10.0 MB of transfers, for an inherent compressibility factor of 2.26; the `amsterdam` chunk represents 8.46 MB of files and 11.6 MB of transfers, for an inherent compressibility factor of 1.37.

None of our experiments placed any limits on the size of the proxy's cache, since it seems reasonable for a mirror host (or DHT) to store a full copy of a 10–70 GB Web site. In the future, we plan to investigate the impact of the size of the cache on SSL splitting's performance.

6.3.2 Measurements

Ideally, SSL splitting's bandwidth utilization should be close to the inherent compressibility of the input trace. To test this, we played back the two 10 MB trace chunks to a standard HTTP server, a standard HTTPS server, an SSL splitting proxy with a cold cache, and an SSL splitting proxy with a warm cache; in each case, we measured the total number of bytes sent on the server's network interface. The resulting bandwidth usage ratios (measured in bytes of bandwidth used per bytes of file throughput) are shown as the gray bars in Figures 10 and 11. (The other bars are explained later in this section.)

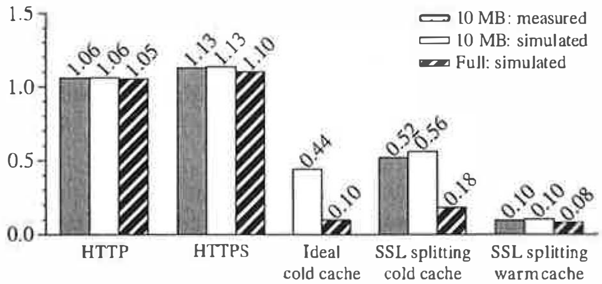


Figure 10: Bandwidth usage: `www.lcs` trace.

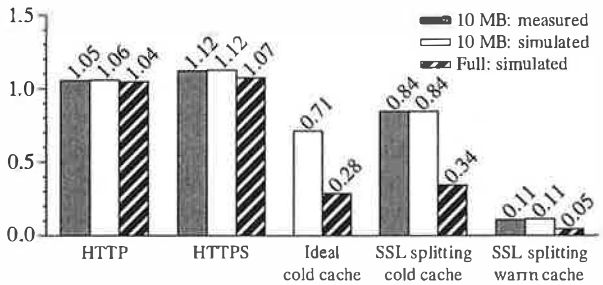


Figure 11: Bandwidth usage: `amsterdam` trace.

As expected, SSL splitting with a cold cache achieves a compression ratio of 2.04 on `www.lcs` and 1.25 on `amsterdam`, with respect to HTTP; the compression is about 5% more with respect to HTTPS, very close to the inherent redundancy. If the cache is warmed before running the trace, the compression ratio is approximately a factor of 11 for `www.lcs` and 10 for `amsterdam`. Analysis of the portions of the trace file preceding the `www.lcs` chunk indicate that if the previous two weeks had been cached by the proxy, the cache would have been approximately 90% warm. The `amsterdam` chunk is too close to the beginning of the trace to perform this analysis.

6.3.3 Simulations

The 10 MB trace chunks used in our experiments span only a few hours each; thus, they might not be representative of Web site traffic over long periods. In addition, the short chunks do not contain as high a degree of repetition as the long traces.

Since it is impractical to replay a several-month-long trace from a busy Web site over an ADSL link, we turned to simulation to estimate the likely performance of SSL splitting over long periods. Based on the data collected in the single-file microbenchmark, we constructed a simple linear model of the performance of HTTP, HTTPS, and SSL splitting in the uncached case: the bandwidth used is a fixed per-file cost plus a marginal per-byte cost. The marginal cost is slightly greater than one because of packet and record overhead.

We model the cost of SSL splitting (in the cached case) as a piecewise linear function: for files smaller than 4,000 bytes, the marginal cost is greater than one, but for larger files, the marginal cost per byte is very small, approximately 1/250. The marginal cost increases to about 1/120 for files larger than 2^{22} bytes.

This simple performance model fits the microbenchmark results to within 5%. In addition, to validate the model, we simulated each experiment with 10 MB trace chunks. The simulated results, shown as the white bars in Figures 10 and 11, agree very closely with the measured results.

The results of simulating the full traces, shown as the striped bars, demonstrates that SSL splitting can take advantage of most of the available redundancy. For example, on the `www.lcs` trace, SSL splitting would have a bandwidth savings of 83% over HTTP; an ideal protocol with zero overhead and perfect caching would save 90%.

6.4 Latency

When the proxy's cache is cold, SSL splitting performs similar work to regular SSL; thus, we expect their latency characteristics to be similar. However, repetition within the trace confuses the analysis of latency factors, since cached files are faster to download than uncached files. To address this issue, we filtered out repetitions from the `www.lcs` trace chunk, and using the resulting uncacheable trace, measured the start and end times of each request. In order to avoid congestion effects, we performed requests one at a time.

The resulting graph of latencies versus file size is shown in Figure 12. It shows three clear lines, one for each of HTTP, HTTPS, and cold SSL splitting. Cold SSL splitting is about 10% faster than HTTPS for small file sizes and about 10% slower than HTTPS for large file sizes, but for the most part they are a close match: the majority of file downloads had less than a 5% difference between the two latencies.

7 Discussion

7.1 The implications of key exposure

The implementation of SSL splitting provides the option to accept clients using a cipher-suite with encryption, and to expose intentionally the encryption key to

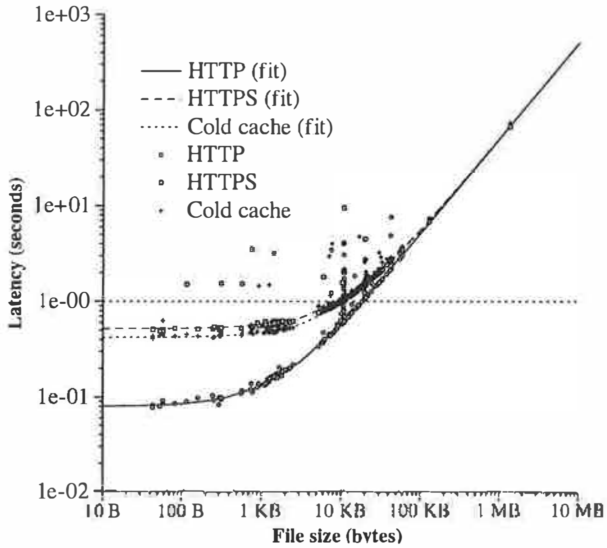


Figure 12: Distribution of latencies vs. file sizes.

the proxy (see Section 3.4). Since the only indication of security in most web browsers is a simple on/off lock icon, there is a legitimate question of whether it is reasonable to mislead clients about whether their communications with the server are encrypted and secure against eavesdroppers. We expect that most applications of SSL splitting will not have any Web forms, since it would be pointless to try to cache dynamic content. However, a user browsing an URL beginning with `https:` might reasonably believe that his browsing pattern was not available to eavesdroppers; the only way to deal with this is to notify the user in the text of the Web page. In any case, normal SSL provides no guarantee that the server will keep its transmissions private; SSL splitting is merely equivalent to an SSL server which sends a carbon copy of all transmissions to a proxy.

7.2 Alternative proxy design

In the current design of the SSL splitting protocol, the cache is a transparent forwarding proxy operating at the level of SSL records. An alternative approach would have been to have the client connect directly to the server, and have the server compress its outgoing stream by converting outgoing IP packets containing SSL records to stub records sent to the cache. The cache would then reconstitute the IP packets and send them to the client, forging the server's IP address and the TCP header. This approach would have the benefit of reducing the number of round-trip packet flight times from four to three, and would also permit the server to "stripe" a connection with a single client across multiple caches. An additional benefit is that no DNS or HTTP redirection technique would be necessary, since the client's transmissions would be directly to the server. However, operating at the IP layer instead of the SSL record layer is fraught with peril: operating system interfaces are nonstandard and unreliable, networks are likely to black-hole forged packets, and TCP will not behave properly. Also, such an approach risks creating an "open relay" which could be used by malicious clients to hide the source of a denial-of-service attack.

8 Related work

Caching and replication in the Web is a subject of much study. Like content-distribution networks [12] and peer-to-peer systems [17], the primary focus of Barnraising is cooperatively sharing the load of serving data. The main difference between Barnraising and previous work is the use of SSL splitting, which allows Barnraising to serve data securely through untrusted proxies to unmodified clients.

8.1 Verifying integrity

The standard approach to providing integrity of data is signing the cryptographic hash of the data with the server's private key (or with the private key of the data's owner). When the client receives the data and its signature (perhaps through different channels), it verifies the integrity of the data by verifying the signature. This solution is typically bundled in the client of a specific application, which users must download to use the application. RPM [25] and FreeNet [2] are among the many applications that use this solution.

The system closest in spirit to Barnraising is read-only SFS [8]. SFSRO allows secure distribution of software bundles through untrusted machines. It provides a generic file system interface, allowing unmodified applications to use SFSRO to distribute data securely. However, SFSRO requires that an SFS client runs on the client machine, which restricts its deployment to SFS users. On the other hand, unlike SSL splitting, the SFSRO server has to serve only the root block to clients, and the computational requirements on the server, untrusted machines, and client are low.

The Secure HTTP (S-HTTP) [20] protocol contains built-in support for caching proxies, in the form of the "320 SHTTP Not Modified" response code. Like SSL splitting, S-HTTP provides an end-to-end freshness and integrity guarantee, but it also provides limited support for confidentiality from the proxy. S-HTTP's computational requirements are similar to SSL, and like SFSRO, the deployment of this protocol is limited.

Untrusted surrogates [6] allow storage-limited clients to cache data on nearby surrogate machines. A server stores data on the surrogate on behalf of the client, and sends the hash of the data to the client; hence, the client can verify the integrity of data when retrieved from the surrogate.

8.2 HTTPS proxies

WASP is a proxy for HTTPS connections [14]. Like SSL splitting, it doesn't require client changes, and defines a separate protocol between proxy and server. Unlike SSL splitting, WASP sends the SSL master secret to the proxy. Since SSL uses the master secret to compute the session keys for both encryption and authentication, this solution puts considerably more trust in the proxy than SSL splitting does. A malicious WASP proxy can change the cached data without the client knowing it.

Proxy certificates [23] provides restricted impersonation within an X.509 public-key infrastructure. A Web site could generate a proxy certificate and hand it to a proxy. The client can then verify the proxy certificate to determine whether the proxy is trusted by the web site to serve the data. Proxy certificates require client changes to process the new X.509 certificate extensions fields.

8.3 Content distribution systems

Commercial content-distribution systems [12] own the machines they use for serving data and therefore trust them. When a client contacts a server with HTTPS via a content-distribution network, the client must trust the content-distribution network to authenticate the server. If SSL splitting were used, the client itself could authenticate the server; also, this would simplify the operation of the content-distribution system.

Most of the content distribution systems based on recently-developed, scalable lookup primitives [21, 13, 3] protect the integrity of data by identifying the data by its cryptographic hash, but the clients must run specialized software to participate in those systems. Squirrel [9] doesn't require special client software, but it doesn't provide data integrity. These systems complement Barnraising by providing it with good techniques for organizing the proxies.

9 Summary

SSL splitting is a novel technique for safely distributing the network load on Web sites to untrusted proxies without requiring modifications to client machines. However, because SSL splitting is effective only at reducing bandwidth consumption when the proxy has access to the plaintext of the connection, it is not appropriate for applications that require confidentiality with respect to the proxy. In addition, SSL splitting incurs a CPU load on the central server due to public-key cryptography operations; it does not address the issue of distributing this load.

The main benefits of SSL splitting are that it provides an end-to-end data-integrity guarantee to unmodified clients, that it reduces the bandwidth consumed by the server, and that it requires only a simple protocol between the server and the proxy. Experiments with a modified OpenSSL library that supports SSL splitting show significant bandwidth savings for files larger than 4,000 bytes: when the data of a file is cached on the proxy, the server need only transmit the SSL handshake messages, HTTP header, MAC stream, and payload IDs. Because of these advantages and the ease of deployment, we hope that SSL splitting will form a convenient transition path for content-distribution systems to provide end-to-end data integrity.

10 Acknowledgements

We thank David Anderson, Russ Cox, Kevin Fu, Thomer Gil, Jacob Strauss, Richard Tibbetts, the anonymous reviewers, the members of the MIT SIPB, and the members of the PDOS group at MIT. Also, thanks to the denizens of TOE, Noah Meyerhans, and the `www.lcs.mit.edu` webmasters, for making our measurements possible.

More information on SSL splitting and Barnraising can be found at `http://pdos.lcs.mit.edu/barnraising/`.

References

[1] CAIN, B., BARBIR, A., NAIR, R., AND SPATSCHECK, O. Known CN request-routing mechanisms. draft-ietf-cdi-known-request-routing-02.txt, Network Working Group, November 2002.

[2] CLARKE, I., SANDBERG, O., WILEY, B., AND HONG, T. W. Freenet: A distributed anonymous information storage and retrieval system. In *Proc. ICSI Workshop on Design Issues in Anonymity and Unobservability* (Berkeley, California, June 2000). `http://freenet.sourceforge.net`.

[3] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area cooperative storage with CFS. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP '01)* (Oct. 2001).

[4] DIERKS, T., AND RESCORLA, E. The TLS protocol version 1.1. draft-ietf-tls-rfc2246-bis-04.txt, Network Working Group, April 2003.

[5] FIPS 180-1. *Secure Hash Standard.* U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, VA, April 1995.

[6] FLINN, J., SINNAMAHIDEE, S., AND SATYANARAYANAN, M. Data staging on untrusted surrogates. Tech. Rep. IRP-TR-02-2, Intel Research, May 2002.

[7] FREIER, A. O., KARLTON, P., AND KOCHER, P. C. The SSL protocol version 3.0. Internet draft (draft-freier-ssl-version3-02.txt), Network Working Group, November 1996. Work in progress.

[8] FU, K., KAASHOEK, M. F., AND MAZIÈRES, D. Fast and secure distributed read-only file system. *ACM Transactions on Computer Systems 20*, 1 (February 2002), 1–24.

[9] IYER, S., ROWSTRON, A., AND DRUSCHEL, P. Squirrel: A decentralized, peer-to-peer web cache. In *21st ACM Symposium on Principles of Distributed Computing (PODC 2002)* (July 2002).

[10] JANNOTTI, J., GIFFORD, D. K., JOHNSON, K. L., KAASHOEK, M. F., AND O'TOOLE, JR., J. W. Overcast: Reliable multicasting with an overlay network. In *Proc. of the 4th OSDI* (Oct. 2002), pp. 197–212.

[11] KARGER, D., LEIGHTON, T., LEWIN, D., AND SHERMAN, A. Web caching with consistent hashing. In *The eighth Word Wide Web Conference* (Toronto, Canada, May 1999).

[12] KRISHNAMURTHY, B., WILLS, C., AND ZHANG, Y. On the use and performance of content distribution networks. Tech. Rep. TD-52AMHL, ATT Research Labs, Aug. 2001.

[13] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., CZERWINSKI, S., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. OceanStore: An architecture for global-scale persistent storage. In *Proceeedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)* (Boston, MA, November 2000), pp. 190–201.

[14] MODADUGU, N., AND GOH, E.-J. The design and implementation of WASP: a wide-area secure proxy. Tech. rep., Stanford, Oct. 2002. `http://crypto.stanford.edu/~eujin/papers/wasp.ps`.

[15] MOORE, D. *MyDNS.* `http://mydns.bboy.net/`.

[16] MYSQL AB. *MySQL database server.* `http://www.mysql.com/`.

[17] ORAM, A., Ed. *Peer-to-peer: Harnessing the power of disruptive technologies.* O'Reilly, Mar. 2001.

[18] RESCORLA, E. HTTP over TLS. RFC 2818, Network Working Group, May 2000.

[19] RESCORLA, E. *SSL and TLS.* Addison-Wesley, 2001.

[20] RESCORLA, E., AND SCHIFFMAN, A. The Secure HyperText Transfer Protocol. RFC 2660, Network Working Group, 1999.

[21] ROWSTRON, A., AND DRUSCHEL, P. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP '01)* (Oct. 2001).

[22] SHACHAM, H., AND BONEH, D. Fast-track session establishment for TLS. In *Proceedings of NDSS* (Feb. 2002), M. Tripunitara, Ed., Internet Society, pp. 195–202. `http://hovav.net/`.

[23] TUCKE, S., ENGERT, D., FOSTER, I., WELCH, V., THOMPSON, M., PEARLMAN, L., AND KESSELMAN, C. Internet x.509 public key infrastructure proxy certificate profile. Internet draft (draft-ietf-pkix-proxy-03), Network Working Group, October 2002. Work in progress.

[24] WESSELS, D. *Squid internet object cache.* `http://squid.nlanr.net/Squid/`.

[25] WWW.RPM.ORG. *RPM software packaging tool.* `http://www.rpm.org/`.

A New Two-Server Approach for Authentication with Short Secrets

John Brainard, Ari Juels, Burt Kaliski, and Michael Szydlo
RSA Laboratories
Bedford, MA 01730, USA
E-mail: {jbrainard,ajuels,bkaliski,mszydlo}@rsasecurity.com

Abstract

Passwords and PINs continue to remain the most widespread forms of user authentication, despite growing awareness of their security limitations. This is because short secrets are convenient, particularly for an increasingly mobile user population. Many users are interested in employing a variety of computing devices with different forms of connectivity and different software platforms. Such users often find it convenient to authenticate by means of passwords and short secrets, to recover lost passwords by answering personal or "life" questions, and to make similar use of relatively weak secrets.

In typical authentication methods based on short secrets, the secrets (or related values) are stored in a central database. Often overlooked is the vulnerability of the secrets to theft en bloc in the event of server compromise. With this in mind, Ford and Kaliski and others have proposed various password "hardening" schemes involving multiple servers, with password privacy assured provided that some servers remain uncompromised.

In this paper, we describe a new, two-server secure roaming system that benefits from an especially lightweight new set of protocols. In contrast to previous ideas, ours can be implemented so as to require essentially no intensive cryptographic computation by clients. This and other design features render the system, in our view, the most practical proposal to date in this area. We describe in this paper the protocol and implementation challenges and the design choices underlying the system.

1 Introduction

In this paper, we consider a basic, pandemic security problem: How is it possible to provide secure services to users who can authenticate using only short secrets or weak passwords?

This problem is of growing importance as Internet-enabled computing devices become ever more prevalent and versatile. These devices now include among their ranks an abundant variety of mobile phones, personal digital assistants (PDAs), and game consoles, as well as laptop and desktop PCs. The availability of networks of computers to highly mobile user populations, as in corporate environments, means that a single user may regularly employ many different points of remote access. The roaming user may additionally employ any of a number of different devices, not all of which necessarily possess the same software or configuration.

While smartcards and similar key-storage devices offer a secured, harmonized approach to authentication for the roaming user, they lack an adequately developed supporting infrastructure in many computing environments. At present, for example, very few computing devices contain smartcard readers – particularly in the United States. Furthermore, many users find physical authentication tokens inconvenient. Another point militating against a critical reliance on hardware tokens is the common need to authenticate roaming users who have lost or forgotten their tokens, or whose tokens have malfunctioned. Today, this is usually achieved by asking users to provide answers to a set of "life" questions, i.e., questions regarding personal and private information. These observations stress that roaming users must be able to employ passwords or other short pieces of memorable information as a form of authentication. Indeed, short secrets like passwords and answers to life questions are the predominant form of authentication for most users today. They are the focus of our work here.

To ensure usability by a large user population, it is important that passwords be memorable. As a result, those used in practice are often highly vulnerable to brute-force guessing attacks [21]. Good credential-server designs must therefore permit secure authentication assuming a weak key (password) on the part of the user.

1.1 SPAKA protocols

A basic tool for mutual authentication via passwords, and one well developed in the literature, is secure password-authenticated key agreement (SPAKA). Most SPAKA protocols are descendants of Bellovin and Merrit's EKE protocol [3, 4], and are predicated on either Diffie-Hellman key agreement or key agreement using RSA. The client and server share a password, which is used to achieve mutual assurance that a cryptographically strong session key is established privately by the two parties. To address the problem of weak passwords, SPAKA protocols are constructed so as to leak no password information, even in the presence of an active attacker. When used as a means of authentication to obtain credentials from a trusted server, a SPAKA protocol is typically supplemented with a throttling or lockout mechanism to prevent on-line guessing attacks. Many roaming-credentials proposals involve use of a SPAKA protocol as a leverage point for obtaining credentials, or as a freestanding authentication protocol. A comprehensive, current bibliography of research papers on the topic of SPAKA protocols (of which there are dozens) is maintained by David Jablon, and may be found at [17].

The design of most SPAKA protocols overlooks a fundamental problem: The server itself represents a serious vulnerability. As SPAKA protocols require the verifying server to have cleartext access to user passwords (or to derivative material), compromise of the server leads potentially to exposure of the full database of passwords. While many SPAKA protocols store passwords in combination with salt or in some exponentiated form, an attacker who compromises the server still has the possibility of mounting off-line dictionary attacks. Additionally, these systems offer no resistance to server corruption. An attacker that gains control of the authenticating server can spoof successful login attempts.

To address this problem, Ford and Kaliski [13] introduced a system in which passwords are effectively protected through distribution of trust across multiple servers. Mackenzie, Shrimpton, and Jakobsson [24] extended this system, leading to more complex protocols, but with rigorous security reductions in a broadly inclusive attack model. Our work in this paper may be regarded as a complement, rather than a successor to the work of these authors. We propose a rather different technical approach, and also achieve some special benefits in our constructions, such as a substantially reduced computational load on the client. At the same time, we consider a different, and in our view more pragmatic security model than that of other distributed SPAKA protocols.

1.2 Previous work

The scheme of Ford and Kaliski reduces server vulnerability to password leakage by means of a mechanism called password hardening. In their system, a client parlays a weak password into a strong one through interaction with one or multiple hardening servers, each one of which blindly transforms the password using a server secret. Ford and Kaliski describe several ways of doing this. Roughly speaking, the client in their protocol obtains what may be regarded as a blind function evaluation σ_i of its password P from each hardening server S_i. (The function in question is based on a secret unique to each server and user account.) The client combines the set of shares $\{\sigma_i\}$ into a single secret σ, a strong key that the user may then use to decrypt credentials, authenticate herself, etc. Given an appropriate choice of blind function evaluation scheme, servers in this protocol may learn no information, in an information-theoretic sense, about the password P. An additional element of the protocol involves the user authenticating by means of σ (or a key derived from it) to each of the servers, thereby proving successful hardening. The harderened password σ is then employed to decrypt downloaded credentials or authenticate to other servers. We note that the Ford-Kaliski system is designed for credential download, and not password recovery; our system is specially designed to support both. Another important distinction is that in the Ford-Kaliski system, the client interacts with both servers directly. As we describe, an important feature of our proposed system is the configuration of one server in the back-end, yielding stronger privacy protection for users.

Mackenzie et al. extend the system of Ford and Kaliski to a threshold setting. In particular, they demonstrate a protocol such that a client communicating with any k out of n servers can establish session keys with each by means of password-based authentication; even if k 1 servers conspire, the password of the client remains private. Their system can be straightforwardly leveraged to achieve secure downloadable credentials. The Mackenzie et al. system, however, imposes considerable overhead of several types. First, servers must possess a shared global key and local keys as well (for a total of $4n+1$ public keys). The client, additionally, must store $n+1$ (certified) public keys. The client must perform several modular exponentiations per server for each session, while the computational load on the servers is high

as well. Finally, the Mackenzie et al. protocol is somewhat complex, both conceptually and in terms of implementation. On the other hand, the protocol is the first such provided with a rigorous proof of security under the Decision Diffie-Hellman assumption [7] in the random oracle model [2].

Frykholm and Juels [15] adopt a rather different approach, in which encrypted user credentials are stored on a single server. In this system, no trust in the server is required to assure user privacy under appropriate cryptographic assumptions. Roughly stated, user credentials are encrypted under a collection of short passwords or keys. Typically, these are answers to life questions. While the Frykholm-Juels system provides error tolerance, allowing the user to answer some questions incorrectly, it is somewhat impractical for a general population of users, as it requires use of a large number of questions. Indeed, the authors recommend a suite of as many as fifteen such questions to achieve strong security. The work of Frykholm and Juels is an improvement on that of Ellison et al. [11], which was found to have a serious security vulnerability [5]. This approach may be thought of as an extension to that of protecting credentials with password-based encryption. The most common basis for this in practice is the PKCS #5 standard [1].

1.3 Our work: a new, lightweight system

It is our view that most SPAKA protocols are over-engineered for real-world security environments. In particular, we take the position that that mutual authentication is often not a requirement for roaming security protocols per se. Internet security is already heavily dependent upon a trust model involving existing forms of server-side authentication, particularly the well studied Secure Sockets Layer protocol (SSL) [14]. SSL is present in nearly all existing Web browsers. Provided that a browser verifies correct binding between URLs and server-side certificates, as most browsers do, the user achieves a high degree of assurance of the identity of the server with which she has initiated a given session. In other words, server authentication is certainly important, but need not be provided by the same secret as user authentication. Thus many SPAKA protocols may be viewed as replicating functionality already provided in an adequately strong form by SSL, rather than building on such functionality.

Moreover, it may be argued that SPAKA protocols carry a hidden assumption of trust in SSL or similar mechanisms to begin with. SPAKA protocols require the availability of special-purpose software on the client side. Given that a mobile user cannot be certain of the (correct) installation of such software on her device, and that out-of-band distribution of special-purpose software is rare, it is likely that a user will need to download the SPAKA software itself from a trusted source. This argues an a priori requirement for user trust in the identity of a security server via SSL or a related mechanism. In this paper, we assume that the client has a pre-existing mechanism for establishing private channels with server-side authentication, such as SSL.

Our system represents an alternative to SPAKAs in addressing "hardening" problem; it is a two-server solution that is especially simple and practical. The idea is roughly as follows. The client splits a user's password (or other short key) P into shares for the two servers. On presenting a password P' for authentication, the client provides the two servers with a new, random sharing of P'. The servers then compare the two sharings of P and P' in such a way that they learn whether $P = P'$, but no additional information. The client machine of the user need have no involvement in this comparison process.

As we explain, it is beneficial to configure our system such that users interact with only one server on the front-end, and pass messages to a second, back-end server via a protected tunnel. This permits the second server to reference accounts by way of pseudonyms, and thereby furnishes users with an extra level of privacy. Such privacy is particularly valuable in the case where the back-end server is externally administered, as by a security-services organization. Much of our protocol design centers on the management of pseudonyms and on protection against the attacks that naïve use of pseudonyms might give rise to.

1.4 Organization

In section 2, we describe the core cryptographic protocol our system for two-server comparison of secret-shared values. We provide an overview of our architecture in section 3, discussing the security motivations behind our choices. In section 4, we describe two specialized protocols in our system; these are aimed at preventing false-identifier and replay attacks. We provide some implementation details for our system in section 5. We conclude in section 6 with a brief discussion of some future directions.

2 An Equality-Testing Protocol

Let us first reiterate and expand on the intuition behind the core cryptographic algorithm in our system, which we refer to as equality testing. The basic idea is for the user to register her password P by providing random shares to the two servers. On presenting her password during login, she splits her password into shares in a different, random way. The two servers compare the two sharings using a protocol that determines whether the new sharing specifies the same password as the original sharing, without leaking any additional information (even if one server tries to cheat). For convenience, we label the two servers "Blue" and "Red". Where appropriate in subscripts, we use the lower-case labels "blue" and "red".

Registration: Let $\mathcal{H}$ be a large group (of, say, 160-bit order), and $+$ be the group operator. Let f be a collision-free hash function $f : \{0,1\}^* \rightarrow \mathcal{H}$. To share her password at registration, the user selects a random group element $R \in_U \mathcal{H}$. She computes the share P_{blue} for Blue as $P_{blue} = f(P) + R$, while the share P_{red} of Red is simply R. Observe that the share of either server individually provides no information about P.

Authentication: When the user furnishes password P' to authenticate herself, she computes a sharing based on a new random group element $R' \in_U \mathcal{H}$. In this sharing, the values $P'_{blue} = f(P') + R'$ and $P'_{red} = R'$ are sent to Blue and Red respectively.

The servers combine the shares provided during registration with those for authentication very simply as follows. Blue computes $Q_{blue} = P_{blue} \quad P'_{blue} = (f(P) \quad f(P')) + (R \quad R')$, while Red similarly computes $Q_{red} = P_{red} \quad P'_{red} = R \quad R'$. Observe that if $P = P'$, i.e., if the user has provided the correct password, then $f(P)$ and $f(P')$ cancel, so that $Q_{blue} = Q_{red}$. Otherwise, if the user provides $P \neq P'$, the result is that $Q_{blue} \neq Q_{red}$ (barring a collision in f). Thus, to test the user password submitted for authentication, the two servers need merely test whether $Q_{blue} = Q_{red}$, preferably without revealing any additional information.

For this task of equality testing, we require a second, large group $\mathcal{G}$ of order q, for which we let multiplication denote the group operation. The group $\mathcal{G}$ should be one over which the discrete logarithm problem is hard. We assume that the two servers have agreed upon this group in advance, and also have agreed upon (and verified) a generator g for $\mathcal{G}$. We also require a collision-free mapping $w : \mathcal{H} \rightarrow \mathcal{G}$. For equality testing of the values Q_{red} and Q_{blue}, the idea is for the two servers to perform a variant of Diffie-Hellman key exchange. In this variant, however, the values Q_{red} and Q_{blue} are "masked" by the Diffie-Hellman keys. The resulting protocol is inspired by and may be thought of as a technical simplification of the PET protocol in [18]. Our protocol uses only one component of an El Gamal ciphertext [16], instead of the usual pair of components as in PET. Our protocol also shares similarities with SPAKA protocols such as EKE. Indeed, one may think of the equality $Q_{red} = Q_{blue}$ as resulting in a shared secret key, and inequality as yielding different keys for the two servers.

There are two basic differences, however, between the goal of a SPAKA protocol and the equality-testing protocol in our system. A SPAKA protocol, as already noted, is designed for security over a potentially unauthenticated channel. In contrast, our intention is to operate over a private, mutually authenticated channel between the two servers. Moreover, we do not seek to derive a shared key from the protocol execution, but merely to test equality of two secret values with a minimum of information leakage. Our desired task of equality testing in our system is known to cryptographers as the socialist millionaires' problem. (The name derives from the idea that two millionaires wish to know whether they enjoy equal financial standing, but do not wish to reveal additional information to one another.) Several approaches to the socialist millionaires' problem are described in the literature, e.g., [8, 12, 19]. In most of this work, researchers are concerned in addressing the problem to ensure the property of fairness, namely that both parties should learn the answer or neither. We do not consider this issue here, as it does not have a major impact on the overall system design. (A protocol unfairly terminated by one server in our system is no worse than a password guess initiated by an adversary, and may be immediately detected by the other server.) By designing a version of the socialist millionaires' protocol without fairness, moreover, we are able to achieve much better efficiency than these previous solutions, which at best require a number of exponentiations linear in the bit-length of the compared values. Our protocol effectively involves only constant overhead. It is more efficient than the protocol in [18], the only other solution to the socialist millionaires' problem that we know of in the literature with constant overhead.

Note that in this protocol, the client need perform no cryptographic computation, but just a sin-

gle (addition) operation in $\mathcal{H}$. (The client performs some cryptographic computation to establish secure connections with Blue and Red in our system, but this may occur via low-exponent RSA encryption – as in SSL – and thus involves just a small number of modular multiplications.) Moreover, once the client has submitted a sharing, it need have no further involvement in the authentication process. Red and Blue together decide on the correctness of the password submitted for authentication. Given a successful authentication, they can then perform any of a range of functions providing privileges for the user: Each server can send a share of a key for decrypting the user's downloadable credentials, or two servers can jointly issue a signed assertion that the user has authenticated, etc.

2.1 Protocol details

As we have already described the simple sharing protocols employed by the client in our system for registration and authentication, we present in detail only the protocol used by the servers to test the equality $Q_{red} = Q_{blue}$. We assume a private, mutually authenticated channel between the two servers. Should the initiating server (Blue) try to establish multiple, concurrent authentication sessions for a given user account, the other server (Red) will refuse. (In particular, in Figure 1, Red will reject the initiation of a session in which the first flow specifies the same user account as for a previously established, active authentication session.) Alternative approaches permitting concurrent login requests for a single account are possible, but more complicated. If Blue initiates an authentication request with Red for a user U for which Red has received no corresponding authentication request from the user, then Red, after some appropriate delay, will reject the authentication.

Let $Q_{blue,U}$ denote the current share combination that Blue wishes to test for user U, and $Q_{red,U}$ the analogous Red-server share combination for user U. In this and any subsequently described protocols in this paper, if a server fails to validate any mathematical relation denoted by $\stackrel{?}{=}$, $\stackrel{?}{\neq}$, $\stackrel{?}{>}$, or $\stackrel{?}{\in}$, it determines that a protocol failure has taken place; in this case, the authentication session is terminated and the corresponding authentication request rejected.

We let $\in_R$ denote uniform random selection from a set. We indicate by square brackets those computations that Red may perform prior to protocol initiation by Blue, if desired. Our password-equality testing protocol is depicted in Figure 1. We use subscripts *red* or 1 to denote values computed or received by Red and *blue* or 0 for those of Blue. We alternate between these forms of notation for visual clarity. We let h denote a one-way hash function (modeled in our security analysis by a random oracle). In the case where a system may include multiple Blue and/or Red servers, the hash input should include the server identities as well. We let $\|$ denote string concatenation.

For the sake of simplicity, we fix a particular group $\mathcal{G}$ for our protocol description here. In particular, we consider $\mathcal{G}$ to be the prime subgroup of order q in Z_p, for prime $p = 2q + 1$. Use of this particular group is reflected in our protocol by: (1) Use of even exponents e_0 and e_1 to ensure group membership in manipulation of transmitted values, and (2) Membership checks over $\{2, \ldots, p \quad 2\}$. For other choices of group, group membership of manipulated values may be ensured by other means. All arithmetic here is performed mod p.

Implementation choices: A typical choice for p, and that adopted in our system, is a 1024-bit prime. Recall that we select $\mathcal{G}$ to be a subgroup of prime order q for $p = 2q + 1$. For $\mathcal{H}$, we simply select the group consisting of, e.g., 160-bit strings, with XOR as the group operator. We note that a wide variety of other choices is possible. For example, one may achieve greater efficiency by selecting shorter exponents e_0 and e_1, e.g., 160 bits. This yields a system that we hypothesize may be proven secure in the generic model for $\mathcal{G}$, but whose security has not been analyzed in the literature. One might also use smaller subgroups, in which case group-membership testing involves fair computational expense. Alternatively, other choices of group $\mathcal{G}$ may yield higher efficiency. One possibility, for example, is selection of $\mathcal{G}$ as an appropriate group over an elliptic curve. This yields much better efficiency for the exponentiation operations, and also has an efficient test of group membership.

Security: In brief, security in our model states that an adversary with active control of one of the two servers and an arbitrary set of users can do essentially no better in attacking the accounts of honest users than random, on-line guessing. Attacks involving such guessing may be contained by means of standard throttling mechanisms, e.g., shutting down a given account after three incorrect guesses. Of course, our scheme does not offer any robustness against simple server failures. This may be achieved straightforwardly through duplication of the Red and

BLUE server		RED server
$e_0 \in_R \{2, 4, \ldots, q \quad 1\}$ $A = w(Q_{blue,U})$ $Y_0 = Ag^{e_0}$		$[e_1 \in_R \{2, 4, \ldots, q \quad 1\}]$ $[Y_1' = g^{e_1}]$
	$\xrightarrow{Y_0,U}$	
		$B = w(Q_{red,U})$ $Y_1 = BY_1'$ $Z_{red} = (Y_0/B)^{e_1}$ $Z_{red} \overset{?}{\in} \{2, \ldots, p \quad 2\}$ $H_{red} = h(Z_{red} \parallel Y_0 \parallel Y_1 \parallel U)$
	$\xleftarrow{Y_1,H_{red}}$	
$Z_{blue} = (Y_1/A)^{e_0}$ $Z_{blue} \overset{?}{\in} \{2, \ldots, p \quad 2\}$ $H_{blue} = h(Z_{blue} \parallel H_{red})$		
	$\xrightarrow{H_{blue}}$	
$H_{red} \overset{?}{=} h(Z_{blue} \parallel Y_0 \parallel Y_1 \parallel U)$		$H_{blue} \overset{?}{=} h(Z_{red} \parallel H_{red})$

Figure 1: Password-equality testing protocol

Blue servers. We also assume fully private server-authenticated channels between the client and the two servers. In this model, and with the random-oracle assumption [2] on the hash function, we claim that the security of our core cryptographic algorithm for equality testing may be reduced to the computational Diffie-Hellman assumption on the group $\mathcal{G}$.

3 Architectural Motivation and Overview

The security of our equality-testing protocol in our system depends upon the inability of an attacker to compromise both Red and Blue. Heterogeneity in server configurations is thus an important practical security consideration here. At the simplest level, the Red and Blue servers may run different operating systems, thereby broadening the range of technical obstacles confronting the would-be attacker. A further possible step in this direction would be to situate Red and Blue within different organizations, with the hope of minimizing the risk of insider or social-engineering attacks.

The distribution of security across organizations also provides an appealing model of risk management in which legal and financial responsibility for compromise can be flexibly allocated. We can view this as a form of privacy outsourcing, in which one server (say, Blue) is operated by a service provider and the other (say, Red) is operated by what we may refer to as a privacy provider. The privacy provider might be an organization with specialized expertise that is willing to assume the primary burden of security maintenance and likewise to assume a large portion of the legal and financial liability associated with privacy breaches.

For a service provider to adopt this approach in a way appealing to a large and potentially mobile user population, there are two salient requirements:

- Universality: There should be no need for clients to install special-purpose software. In particular, clients should be able to interface with the system by way of standard browser components such as Java and HTML.
- Pseudonymity: Red, i.e., the privacy provider, should be unable to gain explicit access to the user names associated with accounts. At a minimum, clients should be able to interact with this server pseudonymously, i.e., by way of identifiers unlinkable with true account names or IP addresses. This provides a technical obstacle to abuse of account information on the part of the operator of Red. It is also useful to employ pseudonyms in this way so as to limit exposure of account identifiers in case of compromise of

Red.

The requirement of universality in the service-provider model argues that the software in our system, while perhaps installed on some clients as a browser plug-in or standalone executable, should also be available in the form of a Java applet. This applet is dispensed by Blue in our system (although it could be dispensed elsewhere). The applet contains the software to execute our basic two-server protocol, and also contains a public key for Red. This public key serves to establish a private channel from the client to Red via Blue.

Distribution of such applets by Blue raises an immediate concern: Blue might serve bad applets. In particular, an attacker that has compromised Blue in an active fashion can cause that server to distribute applets that contain a false public key for Red – or indeed that do not even run the intended protocol. As we have already explained, the problem of trusted software is present even for SPAKA protocols, given the need of roaming clients to install such software on the fly. Applets or other software may be digitally signed, but most users are unlikely to understand how to employ browser-provided verification tools to check the correctness of the associated code-signing certificate. Rather, we make two observations on this score. First, active compromise of core components of Blue is likely to be much harder than passive compromise. Some hope may be placed in so-called "tripwire" tools that are designed specifically to detect hostile code modification. Additionally, the task of an attacker in compromising Blue in this way is harder than active compromise in traditional cryptographic settings, in the following sense: Any observer can in principle detect the compromise by inspecting applets. Thus, the privacy provider might periodically initiate authentication requests with Blue to monitor its integrity. Another complementary approach is for Red to distribute to interested clients a piece of software that verifies the hash of code served by Blue.

The pseudonymity requirement, particularly the notion that Red should not learn the IP addresses of clients, suggests that the privacy provider should operate Red as a back-end server, i.e., a server that only interacts with other servers, not clients. This is the server-configuration that we adopt in our system. In particular, the client in our system communicates with the Red server via an encrypted tunnel established using the public key for Red. There are in fact several other compelling reasons to operate Red as a back-end server:

- Engineering simplicity: Deployment of Red as a back-end server permits the client to establish a direct connection with only a single server, the normal mode of use for most services on the Internet. A service provider may maintain a single front-end server and treat Red as an external, supporting Web service.
- System isolation: In the outsourcing model, the major burden of liability and security is on Red, and the privacy provider is the primary source of security expertise. Hence it is desirable to isolate Red from open communication on the Internet, restricting its interaction instead to one or more Blue servers exclusively via the protocols in our system, effectively creating a kind of strong, application-layer firewall. This imparts to the system as a whole a higher level of security than if both servers were directly exposed.
- Mitigation of denial-of-service attacks: Isolation of Red as a back-end server is also helpful in minimizing the exposure of Red to denial-of-service attacks, which the operator of Blue, having better familiarity with its own user base, is better equipped to handle.

A serious concern does arise in conjunction with the pseudonymity requirement. Blue must identify a given user name U to Red according to a fixed pseudonym V. One possible attack, then, is for Red to pose as a client authenticating under identifier U, and then see which associated pseudonym V Blue asserts. Red thereby learns the linkage between U and V. There is effectively no good (and practical) way to defend against this type of attack. Instead, we rely on social factors to forestall this such behavior on the part of Red, namely: (1) As the service provider, it is Blue that will hold the list of account names, so that these may be difficult for Red to accumulate en bloc; and (2) Given the risk of damaged reputation, Red should be averse to mounting an attack against pseudonyms. Of course, use of pseudonyms is still beneficial in that passive compromise of Red will not reveal true account identifiers.

4 False Pseudonym and Replay Attacks

Our equality-testing protocol is designed to provide security against corruption of one of the two servers in a single session. Other security problems arise, however, as a result of the use of pseudonyms in our system and also from the need for multiple invocations of the equality-testing protocol. In particular, additional protocols are needed in our system to

defend against what we refer to as false-pseudonym and replay attacks.

4.1 The false-pseudonym problem

The possibility of a massive on-line false-pseudonym attack by a corrupted Blue server represents a serious potential vulnerability. In particular, Blue might create an arbitrarily large set of fictitious accounts on Red under false pseudonyms $\tilde{V}_1, \tilde{V}_2, \ldots$, with a dictionary of passwords of its choice. It can then replay genuine authentication requests for a given user's account against the pseudonyms $\tilde{V}_1, \tilde{V}_2, \ldots$. By repeating replays until it achieves a match, Blue thereby learns the secret information for account U. This attack is particularly serious in that it might proceed indefinitely without detection. Behavior of this kind would not be publicly detectable, in contrast for instance to the attack involving distribution of bad applets.

To address this problem, we require that Blue use a secret, static, one-way function f to map user identifiers to pseudonyms. Blue (in conjunction with the client) then proves to Red for every authentication request that it is asserting the correct pseudonym. One challenge in designing a protocol employing this proof strategy is that the client cannot be permitted to learn the pseudonym for any account until after it has authenticated. Otherwise, Red can learn pseudonyms by posing as a client. A second challenge – as in all of our protocols – is to design a proof protocol that it lightweight for Red, Blue, and especially for the client. We demonstrate a protocol here that requires no intensive cryptographic computation by the client – just a modular inversion and a handful of symmetric-key computations. (With a small modification, the modular inversion can be replaced with a modular multiplication, leading to even lower computational requirements.)

The basis of our protocol is a one-way function of the form $f_x : m \rightarrow m^x$ in a group $\mathcal{G}'$ of order q' over which the Decision Diffie-Hellman problem is hard. This choice of one-way function has two especially desirable features for our protocol construction: (1) It is possible to prove statements about the application of f by employing standard non-interactive zero-knowledge proofs on discrete logarithms; and (2) The function f_x has a multiplicative homomorphism, namely $f_x(a)f_x(b) = f_x(ab)$. Naturally, so as to keep f_x secret, the value x is an integer held privately by Blue. We let g denote a generator and $y = g^x$ denote a corresponding public key distributed to Red.

To render the proof protocol lightweight for the client, we adopt a cut-and-choose proof strategy. The idea is that a client identifier U is represented as a group element in $\mathcal{G}'$. The client computes a random, multiplicative splitting of U over $\mathcal{G}'$ into shares U_0 and U_1; thus $U = U_0U_1$. The client also computes commitments to U_0 and U_1, and transmits these to Red. Blue computes V by application of f_x to each of the shares U_0 and U_1. In particular, Blue sends to Red the values $V_0 = f_x(U_0)$ and $V_1 = f_x(U_1)$. Observe that by the multiplicative homomorphism on f_x, Red can then compute the pseudonym $V = f_x(U) = f_x(U_0)f_x(U_1) = V_0V_1$. To prove that this pseudonym V is the right one, Red sends a random challenge bit b to Blue. Blue then reveals U_b and proves that $V_b = f_x(U_b)$, i.e., that the discrete logarithms $\log_g(y)$ and $\log_{U_b}(V_b)$ are equal. The probability that a cheating Blue is detected in this protocol is 1/2. (More precisely, it is extremely close to 1/2 under the right cryptographic assumptions.) Thus, if Blue attempts to mount a pseudonym attack, say, 80 times, this will be detected by Red with overwhelming probability. Our use of this cut-and-choose protocol, therefore, renders the threat of such an attack by Blue much smaller than the threat of a rogue client that submits password guesses. Meanwhile, Red learns only random, independent shares of U, not U itself. We defer further details of the pseudonym protocol and its integration with the other protocols in our system to the full paper.

4.2 The replay-attack problem

In the case where the client communicates directly with Red and Blue via private channels, an adversary in control of either server does not have the capability of mounting a replay attack, as it has access to only the messages sent by the client to one of the servers. In our implementation here, however, where the client communicates with Red via Blue, this is no longer the case. Indeed, without some additional mechanism to ensure the freshness of the share sent by the client to Red, an adversary in control of Blue can mount a replay attack simply by repeating all communications from the client. While the adversary would not learn the password P this way, she could falsely persuade Red that a successful authentication has just occurred; this would enable the adversary to initiate some joint operation on the user's behalf without the user's presence.

A simple countermeasure is to employ timestamps. In particular, Blue may transmit the current time to the client. Along with its other information,

the client then transmits a MAC of this timestamp under R', the share provided to Red. Provided that Red stores for each user account the latest timestamp accompanying a successful authentication, Red can verify the freshness of a share it receives by verifying that the associated timestamp postdates the latest one stored for the account. A drawback of this approach, however, is the engineering complexity introduced by time-synchronization requirements.

An alternative, therefore, is to employ counters. Blue and Red can maintain for each account a counter logging the number of successful authentication attempts. Blue, then, provides the most recent counter value to the client at the time of authentication, and the client transmits a MAC under R' of this counter as an integrity-protected verifier to be forwarded to Red. Using this verifier, Red can confirm the freshness of the associated authentication request.

The drawback to this type of use of counters is its leakage of account information. An attacker posing as a given user can learn the counter value for the user's account from Blue, and thus gather information about her login patterns. An adversary controlling Red can moreover harvest such counter values without initiating authentication attempts and thus without the risk of alerting Blue to potentially suspicious behavior. By matching these counter values against those stored by Red, such an adversary can correlate pseudonyms with user identities.

It is important, therefore, not to transmit plaintext counter values to clients. Instead, Blue can transmit to an authenticating client a commitment ζ of the counter value γ for the claimed user identity [6, 25]. The client then furnishes to Red (via Blue) a MAC under R' of ζ. On initiating an authentication request, Blue provides to Red the counter value γ and a witness ρ associated with ζ; together, these two pieces of data decommit the associated counter value. In this way, the client does not learn γ, but the integrity of the binding between the counter value γ and a given authentication request is preserved. A hash function represents an efficient way to realize the commitment scheme, and is computationally binding and hiding under the random oracle assumption. In particular, Blue may commit γ as $\zeta = h(\gamma \parallel \rho)$, where the witness ρ is a random bitstring of length l, for an appropriate security parameter l (e.g., $l \geq 160$). To decommit, Blue provides γ and ρ, enabling Red to verify the correctness of ζ. This protocol is depicted in Figure 2. The flows of this protocol are overlaid on those of the full authentication protocol in the our system. Let $\gamma_{blue,U}$ denote the counter value stored for the account of the user U attempting to authenticate and $\gamma_{red,U}$ be the corresponding counter value as stored by Red. At the conclusion of this protocol, on successful authentication by the user, Red sets $\gamma_{red,U} = \gamma_{blue,U}$ and Blue increments $\gamma_{blue,U}$ by one.

5 Implementation

In this section we describe the details of our implementation. In particular we describe how the protocols outlined in this paper are integrated, how the servers are configured, and what the components are of the software programs running on each server.

The goal of the prototype we describe here, following the configuration described in section 3, is to improve the security of a standard Web page login procedure. This prototype augments a Web application on a Blue server with the addition of a special authentication library. While a typical Web site would store or look up a password in its database, the enhanced server makes a function call to this library via an API. In order to fulfill these requests, the library makes requests to the Red server, which acts as a privacy provider.

The first component of the prototype is a small Web site on a Blue server with a user registration and login procedure. The second component is a function library which implements all protocol steps to be executed on Blue. The Web application accesses these functions according to our API. The third component is a Red server, which processes and responds to the requests coming from the Blue server, initiated by our library.

In general terms, the message flow may be understood in terms of the client machine making requests to the Web application on Blue. The Web application makes requests to our library on the same server, which in turn makes requests to the Red server. The client never needs to communicate directly with the the Red server. All messages, including encrypted messages destined for the Red server, are sent via Web requests to the Blue server. This encapsulation makes the user experience transparent; the user is not directly aware of the Red server.

Given that the desired client interface is a standard Web browser, we chose to use HTTP for all message communication. We set up the two servers with the Linux operating system (8.0), including the Apache Web server, configured to support CGI (Common Gateway Interface), and SSL.

Client		BLUE server		RED server
		$\rho \in_R \{0,1\}^l$ $\zeta = h(\gamma_{blue,U} \parallel \rho)$		
	$\xleftarrow{\zeta}$			
$D = MAC_{R'}[\zeta]$				
	$\xrightarrow{D}$		$\xrightarrow{D,\zeta,\rho,\gamma_{blue,U}}$	
				$D \stackrel{?}{=} MAC_{R'}[\zeta]$ $\zeta \stackrel{?}{=} h(\gamma_{blue,U} \parallel \rho)$ $\gamma_{blue,U} \stackrel{?}{>} \gamma_{red,U}$

Figure 2: Replay countermeasure protocol

Using HTTPS automatically provides secure channels between Red and Blue, and between Blue and the client. We note that a different transport mechanism between Red and Blue could have been chosen. However, by formatting Blue's requests to Red as well formatted text messages over HTTP, Red effectively acts as a private "Web service", thereby increasing interoperability and design flexibility.

To serve the Web content and perform authentication protocol, two programs, compiled from C/C++ source code, were installed in the proper Web-server directories. To store permanent and transient user data each of the servers uses an SQL database. Standard libraries are used to interact with the database, to format messages, and to produce HTML.

These building blocks provide the secure online message communication, the data storage, and the basic cryptography needed for a variety of protocols. We now come to the most interesting part, the logic particular to our system. Upon receipt of any message, the main processing function in either Red or Blue checks first that it is a well formed message corresponding to a specific step of our protocol. If so, it executes the protocol step, the components of which are described in this paper.

To complete the description of the prototype we just need to describe how the equality checking protocol, and replay countermeasure protocols are integrated, and how the client makes well formed requests to the Web application on Blue without any addition of software to the Web browser.

This is accomplished as follows. A user wishing to authenticate first obtains from the Blue server an HTML form and a signed Java applet. The form has an input field for the user name and password and hidden fields containing a random salt value and Red's public key. On the client machine, the user enters her user name and password into the input field in the HTML form. When the user clicks the "submit" button, the applet hashes the salt with the password, splits the result into shares, and encrypts one share under Red's public key. The encrypted share, the other share, and a replay-prevention value are formatted into a composite message to be sent to Blue as an HTTP request. Of course, the user does not see this processing, nor the other message components prepared by the applet. The user is just served a confirmation or rejection Web page which indicates whether or not the authentication attempt has succeeded.

We now explain further how the two client requests trigger the remaining protocol steps described in this paper. We first remark that our actual implementation also accommodates authentication via life questions, the approach briefly mentioned in the introduction as an alternate authentication mechanism for users with forgotten passwords or unavailable hardware authentication tokens. This extra functionality entails a few technical details. For one, the Java Applet contains the text of personal questions posed to the user and also contains code to split the multiple answers in parallel. Since the questions may vary by user, the user name may be requested first in a separate form. The system permits decisions regarding the success of authentication via life questions to occur on a threshold basis. For example, an

administrator may configure the system to authenticate users successfully if they answer any three out of five life questions correctly. The system need not reveal to the user which answers are incorrect if the authentication as a whole is unsuccessful. (The servers individually, however, will learn the number which questions were answered correctly.) Another feature worth remarking on is that Red does not learn or store the questions posed to individual users.

In Figure 3, we show how the protocol components for equality testing and replay-prevention are overlaid to form a the composite authentication protocol. For simplicity of presentation, we focus here on the basic case of authentication via a single password, not use of life questions. All messages in this figure use notation consistent with that in Figures 1 and 2. Additionally, we denote message components for the Java applet and final response to the client with *Applet* and $\{PASS/FAIL\}$ and encryption under the Red server's public key by E_{Red}. Since we do not include details of our pseudonym-related protocols in this paper, we omit that part of our protocol from our description here. For brevity, we omit the description of certain secondary details such as data representation, and choice of cryptographic primitives here, but we do note that care must be taken to correctly handle session timeouts and the locking out of a user after too many failed login attempts.

Our prototype implements the client as a moderate-sized Java applet, running to about 2000 source lines. The applet can process a password in about 80 milliseconds on a 700MHz Pentium running Windows XP. Note that this does not include the time required to download and initialize the applet.

The prototype Blue server consists of a set of CGI programs written in C++ and C. The prototype code for the Blue server consists of about 10,000 lines of source, not including the communications and database libraries. The prototype Red server is a Linux application built from about 5,000 lines of C and C++ source code. The Blue and Red servers used in the prototype (two 500 MHz Pentium III systems running SUSE Linux) can verify about 10 passwords per second. The prototype was not optimized for efficiency; we expect that significantly better performance should be possible.

We also remark that a version of the protocol also runs under the Windows operating systems, and that our API is now being implemented as a set of Java classes that may be embedded in Servlets or Enterprise Java Beans. The encapsulation of this functionality within a API is particularly useful, having made its realization language independent, and convenient to integrate with a variety of Web applications.

6 Conclusion

As the protocol designs and prototyping experience presented in this paper demonstrate, our system is a highly practical approach to the problem of secure authentication via weak secrets. By employing two servers, the system is able to offer considerably more protection of sensitive user data than any single-server approach could permit. At the same time, the system architecture avoids many of the conceptual and design complexities of multi-server cryptographic protocols – SPAKA schemes and others – described in the literature.

There are a wealth of other multi-server cryptographic protocols that can doubtless be brought to practical fruition in the two-server framework that our system presents. Some examples include:

- credential download, where encrypted credentials are stored on one server and the decryption key is stored on the other;
- threshold digital signing (see, e.g., [22, 23] for discussion of a special two-party protocol);
- joint authorization (and auditing) of self-service user administration operations such as password reset;
- privacy-preserving information delivery as in, e.g., [9, 10, 20].

Our hope is that our system may serve as a useful springboard for the practical realization of these and related concepts from the security literature.

References

[1] PKCS (Public-Key Cryptography Standard) #5 v2.0, 2002. Available at www.rsasecurity.com/rsalabs/pkcs.

[2] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In 1st ACM Conference on Computer and Communications Security, pages 62–73. ACM Press, 1993.

[3] S. M. Bellovin and M. Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In IEEE Computer Society Symposium on Research in Security and Privacy, pages 72–84. IEEE Press, 1992.

Mes #	Client ←- (SSL) → BLUE	BLUE ←- (SSL) → RED
1	$\xrightarrow{U}$	
2	$\xleftarrow{Applet,\zeta}$	
3	$\xrightarrow{U,f(P')+R',E_{red}(R'),D}$	
4		$\xrightarrow{E_{red}(R'),Y_0,U,D,\zeta,\rho,\gamma_{blue,U}}$
5		$\xleftarrow{Y_1,H_{red}}$
6	$\xrightarrow{H_{blue}}$	
7	$\xleftarrow{\{PASS/FAIL\}}$	

Figure 3: Integrated message flow (without pseudonym protocol)

[4] S. M. Bellovin and M. Merritt. Augmented encrypted key exchange. In 1st ACM Conference on Computer and Communications Security, pages 244–250. ACM Press, 1993.

[5] D. Bleichenbacher and P. Q. Nguyen. Noisy polynomial interpolation and noisy Chinese remaindering. In B. Preneel, editor, EUROCRYPT 2000, pages 53–69. Springer-Verlag, 2000. LNCS no. 1807.

[6] M. Blum. Coin flipping by telephone. In Proceedings of 24th IEEE Compcon, pages 133–137, 1982.

[7] D. Boneh. The Decision Diffie-Hellman problem. In ANTS '98, pages 48–63. Springer-Verlag, 1998. LNCS no. 1423.

[8] F. Boudot, B. Schoenmakers, and J. Traoré. A fair and efficient solution to the socialist millionaires' problem. Discrete Applied Mathematics, 111(1-2):23–36, 2001.

[9] D. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. Communications of the ACM, 24(2):84–88, 1981.

[10] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In IEEE Symposium on Foundations of Computer Science, pages 41–50, 1995.

[11] C. Ellison, C. Hall, R. Milbert, and B. Schneier. Protecting secret keys with personal entropy. Journal of Future Generation Computer Systems, 16(4):311–318, February 2000.

[12] R. Fagin, M. Naor, and P. Winkler. Comparing information without leaking it. CACM, 39(5):77–85, May 1996.

[13] W. Ford and B. S. Kaliski Jr. Server-assisted generation of a strong secret from a password. In Proceedings of the IEEE 9th International Workshop on Enabling Technologies (WETICE). IEEE Press, 2000.

[14] A.O. Freier, P. Karlton, and P.C. Kocher. The SSL protocol version 3.0, November 1996. URL: www.netscape.com/eng/ssl3/draft302.txt.

[15] N. Frykholm and A. Juels. Error-tolerant password recovery. In P. Samarati, editor, 8th ACM Conference on Computer and Communications Security, pages 1–9. ACM Press, 2001.

[16] T. El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. IEEE Transactions on Information Theory, 31:469–472, 1985.

[17] D. P. Jablon. Research papers on strong password authentication, 2002. URL: www.integritysciences.com/links.html.

[18] M. Jakobsson and A. Juels. Mix and match: Secure function evaluation via ciphertexts. In T. Okamoto, editor, ASIACRYPT 2000, pages 162–177. Springer-Verlag, 2000. LNCS no. 1976.

[19] M. Jakobsson and M. Yung. Proving without knowing: On oblivious, agnostic, and blindfolded provers. In CRYPTO '96, pages 186–200, 1996. LNCS no. 1109.

[20] A. Juels. Targeted advertising... and privacy too. In D. Naccache, editor, RSA-CT '01, pages 408–424, 2001. LNCS no. 2020.

[21] Daniel V. Klein. "Foiling the cracker" – A survey of, and improvements to, password security.

In Proceedings of the 2nd USENIX Workshop on Security, pages 5–14, Summer 1990.

[22] P. Mackenzie and M. Reiter. Cryptographic servers for capture-resilient devices. In S. Jajodia, editor, 9th ACM Conference on Computer and Communications Security, pages 10–19. ACM Press, 2001.

[23] P. Mackenzie and M. Reiter. Two-party generation of DSA signatures. In J. Kilian, editor, CRYPTO 2001, pages 137–154. Springer-Verlag, 2001. LNCS no. 2139.

[24] P. Mackenzie, T. Shrimpton, and M. Jakobsson. Threshold password-authenticated key exchange. In M. Yung, editor, CRYPTO 2002, pages 385–400. Springer-Verlag, 2002. LNCS no. 2442.

[25] A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone. Handbook of Applied Cryptography. CRC Press, 1996.

Domain-Based Administration of Identity-Based Cryptosystems for Secure Email and IPSEC

D. K. Smetters and Glenn Durfee
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304
{smetters,gdurfee}@parc.com

Abstract

Effective widespread deployment of cryptographic technologies such as secure email and IPsec has been hampered by the difficulties involved in establishing a large scale public key infrastructure, or PKI. Identity-based cryptography (IBC) can be used to ameliorate some of this problem. However, current approaches to using IBC for email or IPsec require a global, trusted key distribution center. In this paper, we present DNSIBC, a system that captures many of the advantages of using IBC, without requiring a global trust infrastructure. The resulting system can be configured to require almost no user intervention to secure both email and IP-based network traffic. We have built a preliminary implementation of this system in Linux.

1 Introduction

Standards for end-to-end encryption and authentication of email messages and IP-based communication have been in place for several years [8, 18]. Implementations of those standards are provided by most mail clients and network stacks. For the most part, however, we still don't use them. A large reason for this is the difficulty of managing and distributing keys – of having an authentic copy of your desired communication partner's public key when you need it.

Traditional approaches to key distribution in the public key setting rely on a Public Key Infrastructure, or PKI, to authenticate the public keys of users and devices relative to a hierarchical organization of trust. PKIs can be complex and difficult to set up and manage. Even with a PKI, you have solved only part of the problem – it only allows you to determine the authenticity of a user's certified public key once you actually have it. But, that user first has to create a key pair and have it certified, and you still have to obtain a copy of his certificate before you can communicate securely with him.

1.1 Identity-Based Cryptography

Identity-Based Cryptography (IBC), originally introduced by Shamir [26], was proposed as a means to solve this problem. In an identity-based cryptographic scheme, you don't have to obtain your communication partner's public key or certificate from anywhere – you already know it. In such a scheme, your public key is an arbitrary string – e.g. "`smetters@parc.com`" or "`myhost.parc.com`". If that string is chosen to be some identity that your communication partner knows, he can encrypt a message to you using only that string and a set of global system parameters. You decrypt that message using the private key corresponding to your public identity string. That private key is derived from your public identity using the global system parameters and a "*master secret*" – a global system secret held by a trusted third party known as a *Private Key Generator*, or PKG. Given the master secret, the PKG can derive a private key corresponding to any desired public key or identity string. As a result, such a system provides automatic key escrow.

Although identity-based signature schemes have been known for some time (e.g. [11, 12]), practical and secure identity-based encryption (IBE) schemes have been described only recently [2, 6]. One such IBE scheme, that of Boneh and Franklin, is based on the Weil or Tate pairing on supersingular elliptic curves [2]. This work has been used as the basis of several identity-based signature schemes [3, 16, 23], as well as a number of identity-

based key agreement protocols [4, 5, 14, 25, 27]. Several of these protocols allow a pair of communicating parties to agree on a shared symmetric key with no interaction whatsoever [14, 25].

IBC seems like an appealing way to solve the usability problems inherent in traditional approaches to key distribution. Not only do you automatically know the public key of anyone or anything with which you might want to communicate, without having to look it up anywhere, but you can encrypt a message to them before that person has even obtained their private key [2, 26]. If software support makes it easy to do so, a user receiving an encrypted email message seems more likely to obtain the private key necessary to read it than they might be to go through the complex steps necessary to get a digital certificate and provide it to someone who wants to send encrypted email to them (see [28] for a prototype of such a system). Similarly, the noninteractive key exchange protocols available using IBC have been proposed as a means to secure network traffic [1].

Unfortunately, systems that provide the full benefits of traditional IBC – knowing any party's public key with no interaction whatsoever – suffer from tremendous scaling and trust management problems. In order to use your public key in an IBC system, I need to know not only your identity, but also a set of *system parameters* – these include both basic cryptographic parameters like the choice of an elliptic curve, but also includes the public key of the PKG, derived from the PKG's master secret [2]. In an extreme case, using an IBC-based noninteractive key exchange protocol requires not only that I know the system parameters associated with your public key, but that you and I share the same system parameters – and hence we must both trust a single PKG in possession of the corresponding master secret.

In general, traditional approaches to IBC assume that all users will share the same PKG. This means that everyone knows the global system parameters, and hence can immediately derive anyone's public key, but also requires the establishment of a system of global trust, where all users obtain their private keys from the same PKG. The global PKG's master secret can be distributed among several centers using threshold cryptography [2] – if everybody could agree on a set of entities to trust with such a secret. However, the mere existence of such a vulnerable global secret – and the resulting system of global key escrow – is simply unacceptable for most applications. Additionally, a global private key generator would be extremely difficult to make work in practice, as it would have to authenticate the "correct" recipient of each private key in a global system of identifiers.

To address the latter problem, recent work has proposed mechanisms for constructing IBC systems using a hierarchically-organized set of PKGs [14, 17]. Unfortunately, in some of these schemes, PKGs higher in the tree can recover the private keys of PKGs lower in the tree, resulting in a system that has a somewhat easier time of distributing keys, but still requires global trust. Even in schemes without escrow, the tree must still be constructed hierarchically – all the nodes in the tree above yours must be in place before your keys can be generated. The resulting system is even more difficult to implement than a global, hierarchical public key infrastructure – something which so far has been notoriously difficult to establish.

1.2 Our Approach

We would like to take advantage of the usability of IBC without requiring everyone to participate in a global trust model. Our goal is to design a system that balances security and usability in a manner resulting in much wider deployment of secure email and IP security (IPsec). In contrast to the approach of Appenzeller and Lynn [1], we attempt to integrate our approach into existing standards and software, so as to ease deployment. We design our trust model to mirror the trust and management divisions that exist in today's deployed networks. We emphasize autoconfiguration and automatic update as much as possible to minimize practical barriers to use.

Building an IBC system that allows us to manage trust along appropriate boundaries is simple – we merely have each such trust domain run their own Private Key Generator. This means that to communicate securely with a given party, you must know not only that party's identity, but you must obtain the system parameters of their trust domain. While this isn't as simple and seamless as a traditional IBC system, it is considerably simpler than a traditional PKI, as those system parameters are shared by an entire trust domain. At worst, it's equivalent to having to obtain the certificate of the certification authority (CA) serving a given trust domain, and having that, being able to immediately derive the public key of everyone else in that trust domain. It makes the distribution of private keys within a trust domain considerably simpler, as the namespace of identifiers is local to the trust domain, and a smaller population of key recipients needs to be authenticated. Making such a system easy to deploy in practice requires appropriate construction of those trust domains, and the design of software to take advantage of them.

1.2.1 Bootstrapping IBC with Domain-Based Trust

Currently, user identities for email (email addresses), and the identities of hosts (names or IP addresses) are managed at the level of network domains, as described by the Domain Name System, or DNS. DNS delegates management of parts of the Internet name space to individual domains of control. We propose to divide the responsibility for authenticating those email addresses and host identities along exactly the same lines, by having each DNS domain responsible for creating a set of IBC system parameters and distributing private keys to its own users. This is directly analogous to having a given domain run its own Certification Authority and issue certificates to its users and machines.

Recent security extensions to DNS, known as DNSSEC, allow a DNS server to digitally sign the responses to queries, so that they cannot be modified or spoofed [9]. Each DNS server providing DNSSEC services offers up a KEY record for its domain containing its public key, signed by the key of the name server above it in the domain hierarchy. When fully deployed, the DNSSEC hierarchy will terminate in a root key trusted by all DNS clients and servers. In the meantime, parts of the DNSSEC hierarchy can be authenticated using cross-certification. As DNSSEC-capable name servers are already capable of providing and authenticating cryptographic data, they have been suggested as the most practical distribution method for cryptographic keys and certificates to be used by IPsec, TLS, secure email, and other protocols [9, 10, 24]. FreeS/WAN, a standard Linux IPsec distribution, has attempted to use these mechanisms to bootstrap an approach to "opportunistically" encrypt all network traffic, by combining distribution of host IPsec keys in DNS records with records that indicate what machines can act as "security gateways" (IPsec termination points) for machines that cannot terminate IPsec themselves [13].

Using DNSSEC to store and distribute a set of authenticated IBC parameters for a domain, retrievable under the domain entry (e.g. "parc.com") is a simple extension of these approaches. However, we suggest that the resulting combination of IBC and DNSSEC-based parameter distribution has a number of advantages over a traditional PKI, even one that uses the DNS as a key distribution mechanism.

First, it minimizes the amount of information stored in and retrieved from the DNS. IBC parameter information is global to the domain. It is generated once, and only regenerated in the case of master secret compromise. Updating of cryptographic information in the DNS is done once for the domain, with intermittent additional updates (e.g. we use a small amount of transient data, a "salt" to provide revocation of keys through key expiration, see Section 2.1). This can be done under administrator control, and doesn't require either clients or a CA to be able to publish information to the DNS, as would likely be necessary if clients put their certificate or key information into the DNS.

Second, as parameters (and salt) are global to the domain, clients wishing to communicate with multiple parties in the domain (or the same party more than once) must only pull one copy of the domain's system parameters. These it can cache over time, reducing the load on the DNS servers.

Third, individual domains can deploy such a system incrementally – if a domain does not provide parameter information in its DNS entry, it obviously does not participate. And though DNSSEC is some ways from being completely deployed, it is designed to allow trust to be constructed incrementally, subtree by subtree, until the roots are in place. A system that bootstraps trust from DNSSEC will grow naturally along with it.

Fourth, client configuration can be dramatically simplified and automated. Clients don't have to obtain their private key from the domain key distribution center until they need it. In the case of email, this means that the motivation is on the "right foot" – a user having received encrypted email is interested in reading it, and will go and get their private key, while a user who wants to send encrypted email to someone else in a non-IBC system is hard pressed to get that person to go and get themselves a certificate [2]. In the case of IPsec, clients can be designed to auto-configure themselves, and automatically request their private key at installation time. We believe the more setup steps that can proceed independently, the simpler the system will be to deploy in practice.

And finally, the fact that we can simply and automatically generate the public key of any party whose domain participates in the system (and the lack of domain parameters in the DNS will tell us whether they participate) means that we can use simpler cryptographic protocols, and can attempt to automatically secure all of our email and network traffic.

1.2.2 Overview

In the remainder of the paper, we will present a practical system for deploying identity-based cryptography (shown in Figure 1). In Section 2.1, we begin with an

overview of the design issues important in embedding IBC into existing protocols. In Sections 2.3, and 2.2, we describe the components of our system. In Section 3, we describe an email client that uses our approach to secure mail, and in Section 4, we describe how to use this system to secure IPsec traffic. Finally, we present our (in-progress) Linux implementation of this system in Section 5, and finish with related work and conclusions.

2 System Design

We have designed a system for using domain-based trust to implement identity-based cryptography. The overall structure of this system can be seen in Figure 1. Such a system consists of a number of components: first, a set of system parameters and a domain master secret, created by a setup procedure. Second, a private key generator, or PKG, that distributes private keys to authenticated members of the domain. Third, a modified nameserver capable of providing copies of the system parameters to communicating peers. And finally, client software capable of using DNS-based IBC (DNSIBC) to secure communications.

2.1 IBC Setup

The first step in enabling DNSIBC in a domain is the creation of the domain's IBC system parameters (labeled setup in Figure 1). In what follows, we focus on IBC systems implemented using operations over supersingular elliptic curves [2], as there are a large number of encryption, signature, and key exchange protocols than can be used with a single private key pair derived for such a scheme.[1]

To specify the domain's system parameters, we first specify a set of elliptic curve group parameters ("group-params"). These consist of the choice of the curve itself, the field it is defined over, and a generator point, referred to as P. These can be considered as analogous to the group parameters used in standard Diffie-Hellman systems, whether defined over an elliptic curve or over a prime field (where the corresponding choice would be of a prime, p, and a generator, g). Like the group parameters used with Diffie-Hellman schemes, these parameters can be shared by many domains, and sets of such parameters can be predefined by standards bodies for general use. This is done, for example, for Diffie-Hellman parameters for use in IPsec [15], allowing hosts that choose to use the standard groups to simply transmit short identifiers in place of the group parameters.

Once the group-params have been selected, each domain creates its own master-secret, s_d, which is a random value in a range specified by the group-params. The group-parameters and the master-secret are used to derive a corresponding public domain-public-key, s_dP. The master-secret is used later by the PKG to derive the private-key corresponding to any identity string, id, by first converting that id to a point Q_{id} on the curve using a hash function mapToPoint, $Q_{id} = \text{mapToPoint(id)}$, and calculating the private-key as $S_{id} = s_d Q_{id}$.

The resulting domain-params consist of:

domain-params := (group-params, domain-public-key)

The master-secret must be stored securely for use by the PKG, while the domain-params are published publicly using the DNS, as shown in Figure 2.

Revocation To add the ability to revoke identities in this system, we add a form of key expiration [2]. Instead of using the identity id as the public key of a user or host, we use salt||id, where salt is a random string long enough to be unlikely to be chosen at random again (say, ten bytes), and || indicates concatenation. For instance, if your id was `smetters@parc.com`, and the current salt for `parc.com` was `OVQpMJJPpgZn`, your public key for this time period would be `OVQpMJJPpgZnsmetters@parc.com`. The salt is published in the DNS along with the domain-params. When the domain's salt changes, keyholders in the domain know to automatically contact the PKG to update their private keys. By using lifetimes in the DNS (see section 2.2) to control the interval at which we have members of the domain and communicating peers check for an updated salt value, we can control the revocation interval for keys in this system. Because peers will automatically update their cached copy of this domain's system salt, we can easily revoke keys on any schedule with much lower bandwidth requirements than, say, the distribution of Certificate Revocation Lists (CRLs).

2.2 DNS

We extend the Domain Name Service [21, 22] to support publication of the domain-params and salt. We do this by adding two resource record (RR) types to the

[1] Note that in the presentation here and in what follows, we use notation common for the mathematics of elliptic curves – P is a point on a curve, x is an integer drawn from the field over which the curve is defined, and $< xP, yP, xyP >$ is the elliptic curve equivalent of the standard Diffie-Hellman tuple $< g^x, g^y, g^{xy} >$.

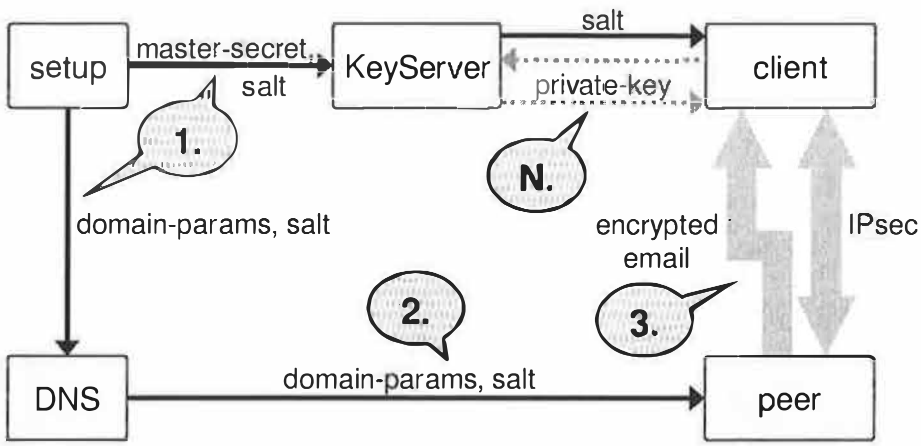


Figure 1: The architecture of our DNSIBC system. The step labeled "N." can take place any time after step "1." Encrypted email is sent normally: i.e., through a standard mail server, without direct interaction between sender and recipient.

DNS. These are PARAMS, which encodes the domain-params, and PSALT, which encodes the salt to be used with these parameters. In this section, we discuss the format and values of the RR fields associated with the new RR types.

The NAME field of the PARAMS and PSALT RRs must be the domain name for which these are the domain-params and salt, respectively. The use of DNS abbreviated names is allowed (although care must be taken so that this does not interact poorly with signing of RRs, discussed below.)

The TTL field of the PARAMS RR should be set to an interval that adequately protects against compromise of the master-secret. In particular, if an unauthorized party obtains the master-secret, she can compute the private-key of any user or machine in the affected domain. If the master-secret is compromised, use of the IBC system should be immediately discontinued until a new master-secret has been generated and corresponding domain-params have propagated to the DNS system. At this point, all previous private-keys and the original master-key are now useless to the attacker. One way to guarantee this recovery happens quickly is to set the TTL field of the PARAMS record to an interval short enough to force frequent updates due to TTL expiration. However, given the typical size of the PARAMS record (approximately 286 bytes with standardized group-params and 690 bytes without), and the unlikelihood of the master-secret compromise (which is presumably on a closely-monitored and well-protected machine accessible from only inside the domain), one can set the PARAMS TTL to a reasonably long period of time (e.g., on the order of days).

The TTL field of the PSALT RR should be set to an interval that protects against compromise of an individual private-key. If a private-key has been compromised, use of IBC on the compromised machine (or user's email account) must be discontinued until a new salt has been generated and propagated to the DNS system. Note that other machines/users may safely continue using IBC-protected IPsec/email. The use of a new salt necessitates (eventual) distribution of a new private-key to every machine/user enrolled in the system. This might place a heavy load on the key server if the update is performed all at once; however, an update could be performed incrementally, as machines discover they are using expired salts, or by having several PSALT records available for a domain used by disjoint blocks of addresses[2]. Since the compromise of an individual private-key is more likely than compromise of the master-secret, and the PSALT RR is significantly shorter than the PARAMS RR, it makes sense to use a much shorter time for the TTL of the PSALT RR, perhaps on the order of several hours. We note that even after the TTL of a PSALT or PARAMS record runs out, it does not necessarily have to be changed. For efficiency, we suggest keeping PARAMS and PSALT records unchanged for long periods of time or until replacement becomes necessary. For the case of the PSALT, this would be until the occurrence of an actual key compromise, or several weeks have passed. The PARAMS could be expected to remain unchanged unless

[2]For example, the PSALT record could be augmented with a field indicating it is to be used for all email address starting with letters a–h. For conciseness, we will not discuss the many interesting variants on the PSALT field that might arise in such a system.

```
struct group_params {
    big_int p, q;
    point_Fp P;
};

struct domain_params {
    struct group_params gp;
    point_Fp domain_public_key;
};

struct DNS_PARAMS_RR {
    unsigned char *NAME;
    uint16      TYPE, CLASS;
    uint32      TTL;
    uint16      RDLENGTH;
    uint16      flags;      // RDATA start
    uint8       protocol;
    uint8       algorithm;
    struct domain_params *domain_params;
};
```

Figure 2: Internal format of the PARAMS DNS resource record and related types.

the master-secret was compromised.

To simplify implementation, the RDATA (record type-specific) portion of the PARAMS resource record uses the same format as that of the KEY RR [9] (see Figure 4). This record contains fields for key usage `flags`, a `protocol` and an `algorithm` (see Figure 2). We suggest using only a subset of the `flags` possible for the KEY record: bits 0 and 1 (prohibition of use of the domain-params for authentication and confidentiality, respectively), and bit 8 (allowing use of domain-params for IPsec). The `protocol` field, which further specifies how this key can be used, should allow an additional value beyond those available in the KEY record: an index for "IBC". The `algorithm` field indicates which basic IBC system is being used, in this case an index indicating the use of IBC based on supersingular elliptic curves.

One might notice that instead of creating a new RR specifically to house the domain-params, we could have extended the KEY resource record type. However, we feel this would be inappropriate for several reasons. First, DNS servers are only required to handle two KEY RRs associated with a DNS name [9]; presumably one is already used as the DNSSEC signing key for this domain, so spending the other on IBC unnecessarily limits further use of the KEY RR type. Second, we wish to use DNSSEC to verify the integrity of a signature on the PARAMS record; this will be signed by the domain

```
parc.com. IN PSALT 0VQpMJJPpgZn
parc.com. IN PARAMS 256 1 1 MIICrgKBgC2c\
   331fS7BexMEzGkWGYcIBPrIH915TnE6c06Ifg\
   fnZBK1cz/PGrF36Z7n1hrHGFHb0hmmHBZb17a\
   YjEG2+MbxvN801DFE6sihKXw0R1Lkk5DtuD...
```

Figure 3: Example zone file containing PSALT and PARAMS DNS resource records.

KEY, and it would be improper to have one KEY record be used to sign a KEY record at the same level. Third, the domain-params are not a public key, and should not be treated as such (for the purposes of revocation, selecting a TTL, etc.) Placing them in a KEY record encourages confusion in this regard, and is in any case deprecated [19].

The RDATA field of the PSALT RR type consists of simply the bytes that make up the salt.

2.2.1 DNSSEC Records for IBE

In order to bootstrap trust in a domain's IBC domain-params there must be a way to verify the validity of the PARAMS and PSALT RRs retrieved from the DNS. We recommend using DNSSEC [9]. A SIG record should be added for the PARAMS resource record owned by this domain. If present, the SIG record must be verifiable using the domain's (traditional) cryptographic public key, which must be available as a KEY record.

2.3 Private Key Generator

In our system, the Private Key Generator (PKG) is a service that computes an entity's private-key using the groups-params, the master-secret, and the client's identity (which could be an email address or a hostname), and the current salt. The PKG obtains the group-params and master-secret from the output of the setup procedure (see Section 2.1). This setup procedure could be run automatically as part of (PKG) initialization. The PKG then waits for key retrieval requests from clients.

Initial Client Key Retrieval Perhaps the most difficult problem in designing a system to easily deploy IBC is enabling clients to automatically retrieve their keys with sufficient security. It is very easy to have clients automatically retrieve their private keys – when a machine or piece of client software (such as an email program)

```
                                1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3
            0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
           +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
           |             flags             |    protocol   |   algorithm   |
           +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
           |                                                               |
           /                      domain parameters                        /
           /                                                               /
           +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4: The wire format of RDATA for the PARAMS DNS resource record.

first realizes that it doesn't have a necessary private key, *e.g.* the first time an IPsec-enabled host boots, or the first time a mail client receives IBC-encrypted mail for a particular user, it attempts to retrieve that key from the domain's PKG. While it would be very easy to create a system to allow a user to securely retrieve that key, for example through a password-protected web site linked to the domain's user database, we would like to enable automated key retrieval.

A host wishing to automatically retrieve its private-key must first find the domain PKG. This can be done, for example, through the use of a standard configuration file general to the domain, or by using a designated record similar to an MX record in the domain's internal DNS to indicate the host acting as the PKG. Then, the client must create an encrypted and preferably authenticated connection to the PKG – encrypted to protect its private-key from eavesdroppers, and authenticated to make sure that an intermediary is not attempting to hijack its key *en route*. Interestingly, a rogue host attempting to pretend to be the PKG is no more than an nuisance, as the client can verify the authenticity of the private-key it receives simply by encrypting and decrypting a message to itself.

In practice, we simply use SSL/TLS to secure communication between the client and the PKG. The PKG can use a traditional server certificate authenticated through any number of standard means – *e.g.* traditional DNSSEC mechanisms for distributing keys, certificates, or CA keys; an internal domain PKI; an external trusted CA; a modified IBC-based version of SSL, or it could even use a self-signed certificate if the internal infrastructure is deemed sufficiently resistant to spoofing attacks.

More importantly, the PKG must be able to authenticate that the client requesting the key for a certain id is actually the client that should have that id. There are a number of ways of accomplishing that. At one extreme, clients could be equipped with private keys and certificates, which they use to authenticate themselves to the PKG. Clients can also be identified using any secrets they already share with the domain (*e.g.* user passwords, or machine domain credentials). It would also be very simple to design mechanisms that use temporary client passwords or "cookies" provided to clients through user registration or administrator action, that they can use to authenticate themselves to the PKG. More practically, simple mechanisms that seem inherently insecure when used globally may be more than sufficient when used inside a trust domain, especially when combined with the ability to verify over time that the correct client received the correct key. In the case of email, this could mean using the ability of a user to receive mail at a particular address to be an indicator that they really are the valid "owner" of that address. Similarly, if the PKG is only allowed to communicate with hosts within a trusted domain (say, behind a firewall on a trusted piece of the network) that in and of itself may be sufficient to authenticate host identity.

Client Key Update Clients must receive new private-keys when the salt (and rarely, the domain-params) are updated (see Section 2.1). Clients can use DNS TTL values to automatically determine the intervals at which they ought to check for new salt or domain-params values (see Section 2.2). While client key updates can obviously be performed in the same manner as initial client key retrieval, some optimization is possible when only the salt has changed. In such a situation, a client whose old private-key has not been compromised can be sent its new private-key encrypted under its old id, which it can use its old private-key to decrypt. The security of the system is maintained as long as the initial private-key exchange was secure.

We note that clients will want to keep a list of several previous private-keys and associated salts. This is necessary to to decrypt email which was sent before, but not read until after, the domain underwent a domain-params or salt change.

3 Secure Email Client

In this section, we describe our approach to providing secure email using domain-based administration of identity-based cryptography. Suppose a sender Alice wishes to send email to a recipient, say Bob with email address "bob@parc.com". Alice's email client first retrieves from the DNS the `PARAMS` and `PSALT` resource records for `parc.com`. These decode into a `domain-params` and `salt`, respectively. Alice then uses identity-based encryption [2] with the domain-params and "salt||bob@parc.com" as the public key to encrypt a symmetric cipher key. This is in turn used to encrypt (and MAC) the email. Alice sends the encrypted email, along with the salt and a digest of the domain-params used to Bob's mail server (which was presumably learned from a `MX` record in the same DNS query.) Note that Bob is not involved in this process. Indeed, up until the encrypted email is sent, Alice has no need to communicate with any machine in parc.com; the DNS will cache all appropriate parameters until their TTLs run out.

Bob's email client then pulls the encrypted email into his mailbox, checks that the digest of the domain-params matches its current knowledge of the domain-params, and pulls from a private store the private-key corresponding to the salt encoded in the message. If either the domain-params digest fails to match, or no entry exists for the salt used in the message, the client asks the DNS for the latest domain-params and salt to see if it needs an updated private-key. If so, it contacts the PKG for a new private-key (using SSL with encryption and mutual authentication); otherwise, it rejects the email as invalid. It then decrypts the message and presents it to Bob.

4 IPsec Client

We would like to use identity-based cryptography to secure IP-based network traffic. Previous work has suggested the use of non-interactive identity-based key exchange protocols to secure traffic between hosts in the same IBC trust domain [1], but did so by inventing a new set of protocols. We'd prefer to use IBC in a way that works easily with existing standards and software, and supports hosts using different sets of IBC parameters.[3] This means using IBC to secure IPsec [18].

[3]We are primarily interested in hosts using domain-based parameters, but the details of how we incorporate IBC into IPsec should be generic across almost any IBC trust distribution mechanism.

IPsec is an IETF standard protocol providing mechanisms for encrypting and authenticating IP packets. This protection is provided using algorithms and symmetric keys negotiated using the Internet Key Exchange protocol, or IKE [15, 20, 24].[4] Clients using IPsec generally implement IKE in a user-space daemon, which negotiates security associations (SAs) and keys with the corresponding IKE daemon on the hosts with which it wishes to communicate securely. The negotiated SAs and keys are then provided to the IPsec implementation in the network stack, which uses them to secure IP packets.

Using IBC to secure IPsec traffic means describing IBC-based key exchange protocols to be used as part of IKE, and implementing them in the IKE daemon. It does not require modification to the network stack components in the kernel. Luckily, it turns out that IBC can be easily accommodated in the existing IKE protocol, with no change to packet structure or protocol flows.

4.1 Structure of IKE

IKE is defined as a particular instantiation of the Internet Security Association and Key Management Protocol (ISAKMP [20]). This is a very complex family of protocols, designed to provide negotiability of algorithms and parameters, optional identity protection for the participating parties, and a number of authentication options.

IKE is divided into two phases. *Phase 1* is used by two peers to establish a secure, authenticated channel over which to communicate; this is referred to the ISAKMP Security Association (SA). During *Phase 2*, those peers go on to negotiate Security Associations to be used by IPsec or other services. Phase 2 traffic is secured using the symmetric keys agreed on as part of the SA negotiated in Phase 1, and therefore is unchanged in our scheme. As part of authenticating each other during Phase 1, the two parties exchange their "identities", in one of several forms, e.g. fully-qualified (DNS) domain names (FQDN), or IP addresses.

Phase 1 can be accomplished in two ways, described as *"modes"*. *Main mode* provides identity protection for the communicating parties by protecting the identifying information they send to each other under either a key derived from an ephemeral Diffie-Hellman exchange, or a public key that they have previously exchanged. *Aggressive mode* is designed to be more efficient, and in

[4]At this writing, IKEv2 is currently under development. In order to experiment with our approach using current software, we have focused our efforts on IKE. Similar modifications should be possible with IKEv2.

Initiator		**Responder**
x_iP	$\longrightarrow$	
	$\longleftarrow$	x_rP

$$K_i = \hat{e}(S_i,\ x_rP)\hat{e}(Q_r,\ x_i s_{rd} P)$$
$$K_r = \hat{e}(S_r,\ x_iP)\hat{e}(Q_i,\ x_r s_{id} P)$$
$$K = K_i = K_r = \hat{e}(x_rS_i\ +\ x_iS_r,\ P)$$
$$K_{psk} = hash(K,\ x_ix_rP)$$

Symbols are defined as follows:

P, q	subgroup generator and subgroup, part of domain's group-params
x_i	initiator's ephemeral elliptic curve Diffie-Hellman private value, $x_i \in_R \mathbb{Z}_q$
x_iP	initiator's ephemeral elliptic curve Diffie-Hellman public value
x_r, x_rP	analogous values for responder
$s_{id}, P_{id} = s_{id}P$	initiator's domain's master-secret and domain-public-key
$s_{rd}, P_{rd} = s_{rd}P$	responder's domain's master-secret and domain-public-key
Q_i, Q_r	mapToPoint of the initiator and responder's identities, i.e., $Q_i = H(ID_i), Q_r = H(ID_r)$
S_i, S_r	initiator and responder's private keys, $S_i = s_{id}Q_i, S_r = s_{rd}Q_r$
$\hat{e}$	an admissible pairing function, $\hat{e}(\text{point}_1,\ \text{point}_2)$
K_i, K_r	method used by initiator and responder, respectively to compute the shared key
K	resulting shared key computed by both parties
K_{psk}	final shared session key without key escrow

Figure 5: Identity-based key exchange algorithm using different domain-public-keys but the same group-params [5].

general does not provide identity protection – participants' identities are sent in the clear as part of their first exchanges.[5]

Finally, Phase 1 (both main and aggressive modes) can be authenticated using one of four protocols: signature-based authentication, two forms of authentication using public key encryption (both of which do provide identity protection in aggressive mode), and authentication using a pre-shared key.

All components of IKE are designed to support a variety of cryptographic algorithms, key lengths, and parameters. Acceptable choices for these variables are listed by the initiator in *proposal* payloads, as part of the first security association (SA) negotiation message it sends to the responder; the responder replies with the single proposal of its choice. This extensibility allows us to incorporate IBC seamlessly into Phase 1, simply by identity-based algorithms as alternatives in these proposal payloads, as long as they can fit into the flows used by IKE's three authentication protocol types. In the next section, we show how to map domain-based IBC onto each of these authentication protocols.

[5]IKE uses two other "modes" – *quick mode* is what is used to perform Phase 2 key exchanges, and *new group mode* can be used after a Phase 1 exchange to change the cryptographic group used by the participants. As cryptographic groups can be negotiated during Phase 1, we present our discussion of group management in that context, and do not discuss either of these modes further.

4.2 IBC-Based IKE

Using IBC in the signature and public-key based authentication modes for IKE Phase 1 is extremely straightforward. It will work even if the participants come from different domains, using unrelated domain-params to issue private keys. All that is required is to select an identity-based signature algorithm (*e.g.* [3, 16, 23]), and/or an identity-based encryption algorithm (*e.g.* Boneh and Franklin's IBE algorithm [2]). Given the selection of appropriate algorithm identifiers and a fixed format in which to exchange the resulting encryptions and signatures, these can be dropped directly into the standard protocol flows provided by IKE [15].

When using identity-based cryptography, the authenticity of a peer's public key is given merely because they prove possession of the private key corresponding to a given identity (sent by the peer as part of IKE, or known *a priori*), relative to the public system parameters of the domain that they claim to be a part of. As a result, the optional IKE messages provided for the ex-

change of certificates can be omitted. Additionally, all of IKE's key exchange protocols can provide perfect forward secrecy (PFS) by generating session keys not from long-term cryptographic secrets (e.g. IBC private keys), but instead using an optional ephemeral Diffie-Hellman key exchange authenticated by those long-term secrets. Combining perfect forward secrecy with IBC automatically avoids the key escrow facilities present in identity-based systems.

4.2.1 IBC and Pre-shared Keys

If the domain-params of the two parties are related, we have another option. At the limit, if the two parties belong to the same domain (*i.e.* have the same domain-params – the same group-params and domain-public-key), and they know each others' identities *a priori*, they can use noninteractive IBC-based key exchange protocols to establish a shared secret key without sending any messages at all [14, 25]. This approach is appealing (*e.g.* [1]), but only applicable to members of the same security domain, and results in a key that is subject to escrow. In practice, hosts using IKE to establish security associations already have to exchange a number of preliminary messages, *e.g.* nonces for freshness, proposals for choices of algorithms, or keying information for PFS. Therefore, they may not be able to take best advantage of the noninteractive nature of these protocols. Additionally, the hosts involved, the responder in particular, may not know the other's identity *a priori* unless it is available as the IP address in current use, or a hostname available through reverse DNS. While these noninteractive protocols can be slightly more computationally efficient than other approaches to using IBC, the narrow set of circumstances in which they can be used, and the potential difficulty in determining whether those circumstances actually apply, make them less appealing.

If the group-params of the two parties are the same, regardless of whether their domain-public-keys are different, then they can use a key exchange protocol similar to (but slightly less efficient than) the noninteractive protocols described above [4, 5]. This would happen if they belonged to different domains, and those domains used the same choice from among the standard sets of group-params. This protocol is illustrated in Figure 5. The resulting protocol avoids the shortcomings of the noninteractive protocols – it is applicable to hosts from different domains, and does not suffer from key escrow. The resulting protocol is effectively a pre-shared key protocol that uses additional elliptic curve Diffie-Hellman information in the computation of the session key. These additional Diffie-Hellman values are directly analogous to the Diffie-Hellman values used in IKE to provide PFS (and in fact do act here to provide PFS), and can be exchanged in the same key exchange (KE) message that standard Diffie-Hellman values would be. The resulting protocol fits neatly into IKE's pre-shared key authentication method, and is illustrated in Figure 6.

The only limitation of the IKE pre-shared key protocol in general is that the two peers do need to know each other's identities – whether they are using IBC (so they can compute the key) or share a traditional static key (so they know which key to use). That means that either they must use aggressive mode so that the identities are exchanged in the first set of messages, or the initiator must know the identity of the responder, either *a priori* or because the responder's identity is either its IP address or a hostname available through reverse DNS lookup.

To use IBC in PSK mode, both peers must know that they are using the same IBC group-params. To achieve this, they exchange information about the group-params in the proposal payloads they use during SA negotiation to suggest the use of PSK. IKE provides standard mechanisms for exchanging group information in the proposal payloads, which were designed originally to identify the Diffie-Hellman groups used for achieving PFS. The same approach can be used to identify IBC group-params, and supports both the description of arbitrary group-params and the use of short identifiers that indicate the use of commonly used standard sets of group-params. Such standardized sets of parameters are used by most hosts for Diffie-Hellman exchanges in IKE, and we anticipate that such standard group-params would be used by most domains in DNSIBC.

5 Implementation

We have built a preliminary version of this system under Linux. Our implementation takes advantage of extensive DNSSEC support present in both the standard implementation of `bind`, the Unix DNS server program, and in `pluto`, the IKE daemon used by Free S/WAN, the most common IPSEC implementation for Linux.

5.1 IBC Libraries

For our initial implementation we wrote a 100%-Java implementation of the low-level field, elliptic curve, and Tate pairing operations necessary to perform identity-based cryptography. This work is based on the C implementation of identity-based encryption available at [28]. Our Java library is used by our PKG server and email

Initiator		**Responder**
HDR, SA,x_iP, N_i, ID_{ii}	$\longrightarrow$	
	$\longleftarrow$	HDR, SA, x_rP, N_r, ID_{ir}, HASH_R
HDR*, HASH_I	$\longrightarrow$	

Keys and authentication values used in protocol are computed as follows:

$$
\begin{aligned}
\mathrm{SKEYID} &= \mathrm{prf}(K_{psk},\ N_i \,||\, N_r) \\
\mathrm{HASH}_I &= \mathrm{prf}(\mathrm{SKEYID},\ x_iP \,||\, x_rP \,||\, \text{CKY-I} \,||\, \text{CKY-R} \,||\, \mathrm{SA}_i \,||\, \mathrm{ID}_{ii}) \\
\mathrm{HASH}_R &= \mathrm{prf}(\mathrm{SKEYID},\ x_rP \,||\, x_iP \,||\, \text{CKY-R} \,||\, \text{CKY-I} \,||\, \mathrm{SA}_i \,||\, \mathrm{ID}_{ir})
\end{aligned}
$$

Symbols are as in Figure 5, with additional symbols defined as follows:

HDR	ISAKMP header
nx_iP, x_rP	as in Figure 5, sent in an ISAKMP KE payload
HDR*	ISAKMP header, payload encrypted under x_ix_rP (also used in computation of K_{psk})
SA	SA negotiation payload with one or more proposals from initiator, one choice from responder
N_i, N_r	initiator and responder nonces
$\mathrm{HASH}_I, \mathrm{HASH}_R$	initiator and responder hashes
prf(key, msg)	keyed pseudorandom function
CKY-I, CKY-R	initiator and responder cookies, respectively
SA_i	the body of the entire SA payload sent by the initiator

Figure 6: Identity-based version of IKE's pre-shared key authentication protocol. Aggressive mode is illustrated here, main mode is similar.

client. We are working to complete a C port of our library for use in our IKE implementation.

5.2 DNS Support

We have modified a DNSSEC-compliant version of the Unix name server program, BIND, to support the distribution of signed parameter and salt records, as shown in Figure 4. These parameters are initially inserted in the DNS during the `setup` phase of the PKG, and are updated as necessary.

The `PARAMS` resource record type is implemented as a modified `KEY` record, with RR type 44. The `PSALT` resource record type is implemented as a modified text (`TXT`) record type, with RR type 45.

5.3 Private Key Generation Service

In a fully deployed system, there are many ways to implement a PKG that provide different amounts of autoconfiguration and different levels of protection on the domain master secret. In our implementation, we have chosen to maximize ease of use and simplicity of setup, in order to encourage deployment.

Our PKG is a standalone program written in Java. On first configuration of a domain (or re-keying of an existing domain), the PKG runs a `setup` sub-program that allows an administrator to select one of the standard sets of domain parameters (see Section 2.1) or to generate her own. The administrator also indicates how the minimum interval permissible before compromised keys can be revoked; this is controlled by the salt lifetime (see Sections 2.1,2.2). The `setup` program then creates a master secret and initial salt, and stores both these and the system parameters in two files: one appropriate for use by the PKG service, and another suitable for incorporating into a DNS zone file. This latter step could also be implemented using DNS dynamic update.

The PKG service then starts on a machine inside the domain network. It listens for connections on a known port (5599), and secures them using SSL/TLS, using a self-signed certificate (obtained from Java's `keytool`) that was previously distributed to clients. Clients connect to the service to obtain their keys either on first initialization, or on change of salt. Authentication of clients is done using the simple "in-vs-out" determination described in Section 2.3, based on the desired identity (email address or FQDN) provided by the client. Private keys, parameters, and the current salt are returned to clients as XML-encoded data protected by the SSL tunnel. Clients then store their new private keys in the location and manner appropriate to them.

To support salt updating, the current salt is passed either on the command line or in a configuration file. Updating the PKG to issue private-keys derived from the new salt is a simple matter of restarting the PKG.

5.4 Email Client

As a preliminary proof of concept, we have implemented a standalone mail client in Java that can send and receive email encrypted with IBE using domain parameters pulled from a DNS server modified as above.

To send encrypted email, our client uses the `dnsjava` package [29], which we modified to accept and parse the new `PARAMS` and `PSALT` DNS resource records. Our client encrypts the message using our Java IBC libraries, encodes it as an XML string, and sends it to the recipient's mail server using the `javax.mail` package.

Upon receiving the first encrypted email message, our client pulls the current salt and its own private-key from the PKG and stores them in a keystore in the local filesystem. On subsequent encrypted email messages received, the client queries the PKG for the latest salt if the current salt's TTL has expired; if the salt changes, it requests a new private-key. In our implementation, the domain-params are included in the private-key, so there is no need to perform a separate check for changed domain-params.

We note that decryption of email is completely transparent to the user: no interaction whatsoever is required pull keys and decrypt messages.

In future work, we would like to incorporate support for domain-based IBC parameters into the existing IBE-based plugins for Outlook and Eudora [28].

5.5 IKE Client

The basis of our initial implementation of an IBC-enabled IKE daemon is a modified version of `pluto`, the IKE daemon provided as part of FreeS/WAN [13]. This implementation was chosen because FreeS/WAN is widely used under Linux, and already provides extensive support for retrieving public keys from DNSSEC. As `pluto` does not provide complete support for use of public-key encryption to authenticate IKE exchanges, we are concentrating on implementing the IBC signature-based and preshared-key based modes described in Section 4. Again, our modified `pluto` is designed for autoconfiguration, requesting its private keys as necessary from the PKG the first time it runs, and updating them on expiration of the domain's salt.

We use the fully-qualified domain name FQDN to identify IPsec hosts, because it allows us to easily support IBC-based IPsec to hosts that use DHCP to obtain their addresses, even if those hosts are currently roaming outside their home domains (a "road warrior" configuration). For those modes of IKE where the responder does not send the initiator their identity before it is needed to derive their public key (*e.g.* both modes authenticated with public key encryption, and main mode authenticated with pre-shared keys), the initiator must already know the responder's identity, or be able to use reverse DNS on their IP address to determine their identity. This is not an extreme limitation, as you frequently know with whom you are initiating a communication with. If it is an unacceptable limitation, an IP address can be used as a host's id without change to any of the above protocols.

6 Related Work

While the value of identity-based encryption for securing email has been recognized for some time [2], only recently have other uses for IBE, and IBC in general, begun to be explored. Much of this work has been at the level of cryptographic primitives, focusing on identity-based signature schemes ([3, 16, 23]) and key exchange protocols ([5, 25, 27]). While much of the work on identity-based cryptography has focused on the model where there is one global trust infrastructure, and one trusted IBC key generator, more recent work has begun to describe primitives that work with less restrictive trust models. This began with work on hierarchical organizations of IBC trust centers [14, 17], and has continued with the design of protocols which work between users in different trust domains that share some of their mathematical parameters [4, 5]. We have been able to make use of this latter work in our own (see section 4.2.1).

Appenzeller and Lynn [1] have suggested using the non-interactive identity-based key exchange protocols suggested by Sakai et al. and others [14, 25] to secure network traffic. While their work is very much in the spirit of ours, it suffers from a number of critical limitations. First, it will only support communication between hosts in the same IBC trust domain. As we note in the introduction, a global IBC trust system is not a realistic deployment scenario. Second, it is a non-standard, special-purpose protocol. As such, it has not been analyzed in any detail, and has no deployment support. In contrast, our approach is to support communicating peers who do

not belong to the same trust domain, and to enable IPsec to use IBC to secure network traffic. While IPsec itself is not a perfect protocol, it is extremely widely deployed, studied and supported, and is subject continuing improvement. Therefore, the general approach to IBC-enabled IPsec presented here seems the most effective way to leverage IBC's deployment advantages to secure network traffic.

7 Advantages Over Alternative Approaches

We believe that our approach offers a number of advantages over existing methods for key distribution to secure email and network traffic. In particular, we believe that our scheme, with its emphasis on autoconfiguration, makes it simple enough to deploy these technologies that they could begin to see much more widespread use. In this section, we compare DNSIBC to alternative approaches.

7.1 Distribution of Trust

An important feature of DNSIBC is the idea of domain-based trust. This is in contrast to standard IBC approaches requiring the establishment of a global trust system [2, 1], which engenders a tremendous management problem – who gets to manage such a system, even if it is distributed, and how do they authenticate requests for private keys – and results in a facility for global key escrow and key compromise.

Our approach is based on a hierarchical distribution of trust, similar to that used by traditional PKIs and hierarchical IBC schemes [14, 17].[6] In contrast to traditional PKIs, however, our approach links its hierarchical organization directly to that of the DNS, rather than having organizations create PKI nodes as a function of their internal organizational structure. Using a domain-based approach, whether for IBC or even a traditional PKI, has the advantage that the things we are intending to authenticate are email senders and network hosts, whose identities derive directly from domains as structured in the DNS. This approach also dramatically simplifies the task faced by someone outside of a given domain who wishes to communicate securely with someone inside that domain, by making it easy for them to find out whether or not there is a cryptographic trust system in place (e.g. IBC system or PKI) in that domain, and to know where to look for credentials in that trust system.

[6] See Section 1.1 for discussion of the shortcomings of current hierarchical IBC schemes.

A domain-based approach also eases autoconfiguration and system setup, both for the system administrator, and the end user. If basic credentials for securing email and network traffic are organized according to the domain, a simple default implementation of a system to create and manage such credentials can be provided along with the other tools used to manage a domain, much as DNSSEC tools now come along with the name server, `bind`. Domains with more sophisticated security needs and resources can replace these simple implementations with something more complex, but they may be good enough for many domains that currently find themselves unable to set up and manage a PKI from "scratch". A domain-based system which uses DNSSEC to root its trust has the added advantage that it removes yet another energy barrier to deployment. Although it is rolling out slowly, there are very good practical reasons for full deployment of DNSSEC. Trust infrastructures that inherit from DNSSEC (e.g. by using your domain or zone's DNSSEC keys to sign and hence authenticate your domain IBC parameters) can take advantage of this momentum, and are therefore much easier to deploy in practice than setting up yet another trust hierarchy whose organization mirrors that of the DNS.

7.2 Use of Identity-Based Cryptography

We have argued strongly above for the practical advantages of domain-based, standardized trust systems. Why, then should we implement such a system with IBC, rather than say, having each domain directly certify the keys [9] or digital certificates [10] of end-entities with DNSSEC, and distribute them through the DNS? Or perhaps have an LDAP server running which maintains a list To see the advantages of IBC in these situations, it is illustrative to focus on the clients – IBC has its strongest advantages there.

7.2.1 Versus Storing Certificates in DNS

Why use IBC, rather than distributing keys or certificates via the DNS? Distribution of a domain root certificate via the DNS would give us a domain trust model similar to DNSIBC, and would make it easy for clients from different domains to find the trust root for their desired communication partner. We could even, in the extreme, automate a domain's certification authority so that clients (email users and network hosts) could automatically request certificates when they needed them. Such a system is actually currently implemented in Microsoft Windows 2000™ Active Directory-based domains that run a Microsoft Certification Authority [7].

Machines belonging to the domain can be configured to automatically request an IPsec certificate when they first join the domain, and that certificate is stored in Active Directory, which also can be used to store and distribute user email certificates. That particular approach is limited to a particular vendor's client and server software, and limits access to the stored certificates to members of the domain, but it could obviously be generalized.

We suggest a number of reasons why IBC might be a better approach. First of all, it minimizes the number of interacting parties in the system, and in particular, the number of parties that need to update the DNS zone information. Using IBC, domain parameters need to be made available by the DNS, but no per-client information needs to be there. In a certificate-based approach, each client needs to place their certificate information into the DNS. An IBC approach also dramatically reduces the bandwidth required to access peers' credential information. To communicate with any number of peers in a given domain, I only need to obtain that domain's parameters once per revocation interval. I can then communicate securely with any email user in the domain, or any domain host, for which I know an address. I can also cache that information and make it available to a population of querying hosts using standard DNS caching software.

Another advantage of this approach is that using it, I can communicate securely with any host or email user whose address I know – but only those whose addresses I know. If every user in a domain has their email address directly represented in the DNS in the form of their digital certificate, "fishing expeditions" to find user identities or the distribution of hosts become much easier.

And finally, this approach preserves the appealing use model of IBC. I can send encrypted email to a user that has not yet bothered to get the private key necessary to decrypt it, or even perhaps to install the software or plug-in necessary to support IBC. Having received such an encrypted email, that user is then considerably more motivated to perform the necessary steps to decrypt it, after which he will continue to seamlessly participate in the system. Similarly, it becomes possible to support both autoconfiguration of IPsec hosts who can retrieve their own keys as part of their setup process, and seamless IPsec termination by trusted proxies provided by the domain for devices not capable of terminating IPsec on their own.

7.2.2 Versus Dynamic Certificate Generation

Lastly, we might consider using instead a system with dynamic or "lazy" certificate generation. An LDAP certificate server could be set up which, if a user or host already has a certificate, returns it. If not, it generates a key pair, makes the certificate available to the outside world, and keeps the private key to be later transmitted to the user or host.

Our IBE-based approach has several advantages over a system such as this. First, in IBE, the process of generating a user's private key is decoupled from the generation of their public key (which is, of course, just their identity.) This allows us to introduce an "air gap" in between the private key generator and the outside world: the private key generator need only be accessible by users or machines within the domain. In contrast, our hypothetical LDAP certificate server, which is on the outside of a domain's firewall, must maintain connectivity with the private key generator at all times, introducing a possible path for an attacker to the private key repository.

Furthermore, this LDAP certificate server must either validate requests or generate key pairs for every request that is made. For instance, the LDAP server either maintains an up-to-date list of email addresses, which an attacker could then quickly probe; or, it generates key pairs for every requested email address, which opens up vulnerability to denial-of-service by flooding the server with bogus requests. In our IBE-based approach, no such attacks are possible.

Lastly, a major strength of integrating identity-based encryption parameters into the Domain Name Service is the propagation and redundancy the DNS provides via caching. In our hypothetical LDAP system, a single service must be contacted in order to send encrypted email to a user in a domain. In our scheme, the identity-based encryption parameters for that domain propagate through the DNS and can be cached locally.

8 Conclusions

We have presented an approach to protecting email and network traffic using identity-based cryptography and domain-based trust. We think that this system provides a simple and easy way to establish widespread support for secured communication, through its thorough support for autoconfiguration, and identity-based cryptog-

raphy's novel solution to the key distribution problem. We have built an initial implementation of this system in Linux as a proof of concept of its effectiveness and usability.

Acknowledgments

The authors would like to thank the referees for their many helpful comments.

References

[1] G. Appenzeller and B. Lynn. Minimal overhead IP security using identity-based encryption. Submitted for publication, `http://rooster.stanford.edu/~ben/pubs/ipibe.pdf`.

[2] D. Boneh and M. Franklin. Identity-based encryption from the Weil pairing. In *Proc. CRYPTO 01*, pages 213–229. Springer-Verlag, 2001. LNCS 2139.

[3] J. Cha and J. Cheon. An identity-based signature from gap diffie-hellman groups. `http://eprint.iacr.org/2002/018`.

[4] L. Chen, K. Harrison, N. P. Smart, and D. Soldera. Applications of multiple trust authorities in pairing based cryptosystems. In *Proceedings of Infrastructure Security: InfraSec 2002*, pages 260–275. Springer-Verlag, 2002. LNCS 2437.

[5] L. Chen and C. Kudla. Identity based authenticated key agreement from pairings. `http://eprint.iacr.org/2002/184`.

[6] C. Cocks. An identity based encryption scheme based on quadratic residues. In *Cryptography and Coding*, pages 360–363. Springer-Verlag, 2001. LNCS 2260.

[7] J. de Clercq. PKI comes of age. *Windows & .NET Magazine*, pages 47–53, May 2002.

[8] S. Dusse, P. Hoffman, B. Ramsdell, L. Lundblade, and L. Repka. *S/MIME Version 2 Message Specification.* IETF - Network Working Group, The Internet Society, March 1998. RFC 2311.

[9] D. Eastlake. *Domain Name System Security Extensions.* IETF - Network Working Group, The Internet Society, March 1999. RFC 2535.

[10] D. Eastlake and O. Gudmundsson. *Storing Certificates in the Domain Name System (DNS).* IETF - Network Working Group, The Internet Society, March 1999. RFC 2538.

[11] U. Feige, A. Fiat, and A. Shamir. Zero knowledge proofs of identity. *Journal of Cryptology*, 1(2):77–94, 1988.

[12] A. Fiat and A. Shamir. How to prove yourself: practical solutions to identification and signature problems. In A. M. Odlyzko, editor, *Proc. CRYPTO 86*, pages 186–194. Springer, 1987. Lecture Notes in Computer Science No. 263.

[13] Free S/WAN Project. Free S/WAN. `http://www.freeswan.org`.

[14] C. Gentry and A. Silverberg. Hierarchical ID-based cryptography. In *Advances in Cryptology - Asiacrypt 2002.* Springer-Verlag, 2002.

[15] D. Harkins and D. Carrel. *The Internet Key Exchange (IKE).* IETF - Network Working Group, The Internet Society, November 1998. RFC 2409.

[16] F. Hess. Exponent group signature schemes and efficient identity based signature schemes based on pairings. `http://eprint.iacr.org/2002/012`.

[17] J. Horwitz and B. Lynn. Toward hierarchical identity-based encryption. In *Proc. EUROCRYPT 02*, pages 466–481. Springer-Verlag, 2002. LNCS 2332.

[18] S. Kent and R. Atkinson. *Security Architecture for the Internet Protocol.* IETF - Network Working Group, The Internet Society, November 1998. RFC 2401.

[19] D. Massey and S. Rose. *Limiting the Scope of the KEY Resource Record.* IETF - Network Working Group, The Internet Society, December 2002. RFC 3445.

[20] D. Maughan, M. Schertler, M. Schneider, and J. Turner. *Internet Security Association and Key Management Protocol (ISAKMP).* IETF - Network Working Group, The Internet Society, November 1998. RFC 2408.

[21] P. Mockapetris. *Domain Names – Concepts and Facilities.* IETF - Network Working Group, The Internet Society, November 1987. RFC 1034.

[22] P. Mockapetris. *Domain Names – Implementation and Specification.* IETF - Network Working Group, The Internet Society, November 1987. RFC 1035.

[23] K. Paterson. ID-based signatures from pairings on elliptic curves. `http://eprint.iacr.org/2002/004`.

[24] D. Piper. *The Internet IP Security Domain of Interpretation for ISAKMP.* IETF - Network Working Group, The Internet Society, November 1998. RFC 2407.

[25] R. Sakai, K. Ohgishi, and M. Kasahara. Cryptosystems based on pairing. In *Proceedings of the Symposium on Cryptography and Information Security (SCIS 2000)*, Okinawa, Japan, January 2000.

[26] A. Shamir. Identity-based cryptosystems and signature schemes. In G. R. Blakley and D. C. Chaum, editors, *Proc. CRYPTO 84*, pages 47–53. Springer, 1985. Lecture Notes in Computer Science No. 196.

[27] N. Smart. An identity based authenticated key agreement protocol based on the weil pairing. *Electronics Letters*, 38:630–632, 2002.

[28] Stanford Applied Cryptography Group. IBE secure e-mail. `http://crypto.stanford.edu/ibe`.

[29] B. Wellington. dnsjava: An implementation of DNS in Java. `http://www.xbill.org/dnsjava/`.

Preventing Privilege Escalation

Niels Provos
CITI, University of Michigan

Markus Friedl
GeNUA

Peter Honeyman
CITI, University of Michigan

Abstract

Many operating system services require special privilege to execute their tasks. A programming error in a privileged service opens the door to system compromise in the form of unauthorized acquisition of privileges. In the worst case, a remote attacker may obtain superuser privileges. In this paper, we discuss the methodology and design of privilege separation, a generic approach that lets parts of an application run with different levels of privilege. Programming errors occurring in the unprivileged parts can no longer be abused to gain unauthorized privileges. Privilege separation is orthogonal to capability systems or application confinement and enhances the security of such systems even further.

Privilege separation is especially useful for system services that authenticate users. These services execute privileged operations depending on internal state not known to an application confinement mechanism. As a concrete example, the concept of privilege separation has been implemented in OpenSSH. However, privilege separation is equally useful for other authenticating services. We illustrate how separation of privileges reduces the amount of OpenSSH code that is executed with special privilege. Privilege separation prevents known security vulnerabilities in prior OpenSSH versions including some that were unknown at the time of its implementation.

1 Introduction

Services running on computers connected to the Internet present a target for adversaries to compromise their security. This can lead to unauthorized access to sensitive data or resources.

Services that require special privilege for their operation are critically sensitive. A programming error here may allow an adversary to obtain and abuse the special privilege.

The degree of the escalation depends on which privileges the adversary is authorized to hold and which privileges can be obtained in a successful attack. For example, a programming error that permits a user to gain extra privilege after successful authentication limits the degree of escalation because the user is already authorized to hold some privilege. On the other hand, a remote adversary gaining superuser privilege with no authentication presents a greater degree of escalation.

For services that are part of the critical Internet infrastructure is it particularly important to protect against programming errors. Sometimes these services need to retain special privilege throughout their lifetime. For example, in SSH, the SSH daemon needs to know the private host key during re-keying to authenticate the key exchange. The daemon also needs to open new pseudo-terminals when the SSH client so requests. These operations require durable special privilege as they can be requested at any time during the lifetime of a SSH connection. In current SSH implementations, therefore, an exploitable programming error allows an adversary to obtain superuser privilege.

Several approaches to help prevent security problems related to programming errors have been proposed. Among them are type-safe languages [30] and operating system mechanisms such as protection domains [11] or application confinement [18, 21, 28]. However, these solutions do not apply to many existing applications written in C running on generic Unix operating systems. Furthermore, system services that authenticate users are difficult to confine because execution of privileged operations depends on internal state not known to the sandbox.

Instead, this paper discusses the methodology and design of *privilege separation*, a generic approach to limit the scope of programming bugs. The basic principle of privilege separation is to reduce the amount of code that runs with special privilege without affecting or limiting the functionality of the service. This narrows the exposure to bugs in code that is executed with privileges. Ideally, the only consequence of an error in a privilege separated service is denial of service to the adversary himself.

The principle of separating privileges applies to any privileged service on Unix operating systems. It is especially useful for system services that grant authenticated users special privilege. Such services are difficult to confine because the internal state of a service is not

known to an application confinement system and for that reason it cannot restrict operations that the service might perform for authenticated users. As a result, an adversary who gains unauthorized control over the service may execute the same operations as any authenticated user. With privilege separation, the adversary controls only the unprivileged code path and obtains no unauthorized privilege.

Privilege separation also facilitates source code audits by reducing the amount of code that needs to be inspected intensively. While all source code requires auditing, the size of code that is most critical to security decreases.

In Unix, every process runs within its own protection domain, *i.e.*, the operating system protects the address space of a process from manipulation and control by unrelated users. Using this feature, we accomplish privilege separation by spawning unprivileged children from a privileged parent. To execute privileged operations, an unprivileged child asks its privileged parent to execute the operation on behalf of the child. An adversary who gains control over the child is confined in its protection domain and does not gain control over the parent.

In this paper, we use OpenSSH as an example of a service whose privileges can be separated. We show that bugs in OpenSSH that led to system compromise are completely contained by privilege separation. Privilege separation requires small changes to existing code and incurs no noticeable performance penalty.

The rest of the paper is organized as follows. In Section 2, we discuss the principle of least privilege. We introduce the concept of privilege separation in Section 3 and describe a generic implementation for Unix operating system platforms. We explain the implementation of privilege separation in OpenSSH in Section 4. In Section 5, we discuss how privilege separation improves security in OpenSSH. We analyze performance impact in Section 6. Section 7 describes related work. Finally, we conclude in Section 8.

2 Least Privilege

We refer to a *privilege* as a security attribute that is required for certain operations. Privileges are not unique and may be held by multiple entities.

The motivation for this effort is the principle of least privilege: every program and every user should operate using the least amount of privilege necessary to complete the job [23]. Applying the principle to application design limits unintended damage resulting from programming errors. Linden [15] suggests three approaches to application design that help prevent unanticipated consequences from such errors: defensive programming, language enforced protection, and protection mechanisms supported by the operating system.

The latter two approaches are not applicable to many Unix-like operating systems because they are developed in the C language which lacks type-safety or other protection enforcement. Though some systems have started to support non-executable stack pages which prevent many stack overflows from being exploitable, even this simple mechanism is not available for most Unix platforms.

Furthermore, the Unix security model is very coarse grained. Process privileges are organized in a flat tree. At the root of the tree is the superuser. Its leaves are the users of the system. The superuser has access to every process, whereas users may not control processes of other users. Privileges that are related to file system access have finer granularity because the system grants access based on the identity of the user and his group memberships. In general, privileged operations are executed via system calls in the Unix kernel, which differentiates mainly between the superuser and everyone else.

This leaves defensive programming, which attempts to prevent errors by checking the integrity of parameters and data structures at implementation, compile or run time. For example, defensive programming prevents buffer overflows by checking that the buffer is large enough to hold the data that is being copied into it. Improved library interfaces like *strlcpy* and *strlcat* help programmers avoid buffer overflows [17].

Nonetheless, for complex applications it is still inevitable that programming errors remain. Furthermore, even the most carefully written application can be affected by third-party libraries and modules that have not been developed with the same stringency. The likelihood of bugs is high, and an adversary will try to use those bugs to gain special privilege. Even if the principle of least privilege has been followed, an adversary may still gain those privileges that are necessary for the application to operate.

3 Privilege Separation

This section presents an approach called *privilege separation* that cleaves an application into privileged and unprivileged parts. Its philosophy is similar to the decomposition found in micro-kernels or in Unix command line tools. Privilege separation is orthogonal to other protection mechanisms that an operating system might support, *e.g.*, capabilities or protection domains.

We describe an implementation of privilege separation that does not require special support from the operating system kernel and as such may be implemented on almost any Unix-like operating system.

The goal of privilege separation is to reduce the amount of code that runs with special privilege. We achieve this by splitting an application into parts. One part runs with privileges and the others run without them. We call the privileged part the *monitor* and the unprivileged parts the *slaves*. While there is usually only one slave, it is not a requirement. A slave must ask the monitor to perform any operation that requires privileges. Before serving a request from the slave, the monitor first validates it. If the request is currently permitted, the monitor executes it and communicates the results back to the slave.

In order to separate the privileges in a service, it is necessary to identify the operations that require them. The number of such operations is usually small compared to the operations that can be executed without special privilege. Privilege separation reduces the number of programming errors that occur in a privileged code path. Furthermore, source code audits can focus on code that is executed with special privilege, which can further reduce the incidence of unauthorized privilege escalation.

Although errors in the unprivileged code path cannot result in any immediate privilege escalation, it might still be possible to abuse them for other attacks like resource starvation. Such denial of service attacks are beyond the scope of this paper.

In the remainder of this section, we explain the Unix mechanisms that allow us to implement a privilege separated service. Processes are protection domains in a Unix system. That means that one process cannot control another unrelated process. To achieve privilege separation, we create two entities: a privileged parent process that acts as the monitor and an unprivileged child process that acts as the slave. The privileged parent can be modeled by a finite-state machine (FSM) that monitors the progress of the unprivileged child. The parent accepts requests from the child for actions that require privileges. The set of actions that are permitted changes over time and depends on the current state of the FSM. If the number of actions that require privileges is small, most of the application code is executed by the unprivileged child.

The design of the interface is important as it provides a venue of attack for an adversary who manages to compromise the unprivileged child. For example, the interface should not provide mechanisms that allow the export of sensitive information to the the child, like a private signing key. Instead, the interface provides a request that allows the child to request a digital signature.

A privilege separated service can be in one of two phases:

- Pre-Authentication Phase: A user has contacted a system service but is not yet authenticated. In this case, the unprivileged child has no process privileges and no rights to access the file system.

- Post-Authentication Phase: The user has successfully authenticated to the system. The child has the privileges of the user including file system access, but does not hold any other special privilege. However, special privilege are still required to create new pseudo-terminals or to perform other privileged operations. For those operations, the child must request an action from the privileged parent.

The unprivileged child is created by changing its user identification (UID) and group identification (GID) to otherwise unused IDs. This is achieved by first starting a privileged monitor process. It forks a slave process. To prevent access to the file system, the child changes the root of its file system to an empty directory in which it is not allowed to create any files. Afterwards, the slave changes its UID and GID to lose its process privileges.

To enable slave requests to the monitor, we use inter-process communication (IPC). There are many different ways to allow communication between processes: pipes, shared memory, etc. In our case, we establish a socket between the two processes using the *socketpair* system call. The file descriptor is inherited by the forked child.

A slave may request different types of privileged operations from the monitor. We classify them depending on the result the slave expects to achieve: *Information*, *Capabilities*, or *Change of Identity*.

A child issues an informational request if acquiring the information requires privileges. The request starts with a 32-bit length field followed by an 8-bit number that determines the request type. In general, the monitor checks every request to see if it is allowed. It may also cache the request and result. In the pre-authentication phase, challenge-response authentication can be handled via informational requests. For example, the child first requests a challenge from the privileged monitor. After receiving the challenge, the child presents it to the user and requests authentication from the monitor by presenting the response to it. In this case, the monitor remembers the challenge that it created and verifies that the response matches. The result is either successful or unsuccessful authentication. In OpenSSH, most privileged operations can

```
cmsg = CMSG_FIRSTHDR(&msg);
cmsg->cmsg_len = CMSG_LEN(sizeof(int));
cmsg->cmsg_level = SOL_SOCKET;
cmsg->cmsg_type = SCM_RIGHTS;
*(int *)CMSG_DATA(cmsg) = fd;
```

Figure 1: File descriptor passing enables us to send a file descriptor to another process using a special control message. With file descriptor passing, the monitor can grant an unprivileged child access to a file that the child is not allowed to open itself.

```
mm_master_t *mm_create(mm_master_t *, size_t);
void mm_destroy(mm_master_t *);
void *mm_malloc(mm_master_t *, size_t);
void mm_free(mm_master_t *, void *);
void mm_share_sync(mm_master_t **, mm_master_t **)
```

Figure 2: These functions represent the interface for shared memory allocation. They allow us to export dynamically allocated data from a child process to its parent without changing address space references contained in opaque data objects.

be implemented with informational requests.

Ordinarily, the only capability available to a process in a Unix operating systems is a file descriptor. When a slave requests a capability, it expects to receive a file descriptor from the privileged monitor that it could not obtain itself. A good example of this is a service that provides a pseudo-terminal to an authenticated user. Creating a pseudo-terminal involves opening a device owned by the superuser and changing its ownership to the authenticated user, which requires special privilege.

Modern Unix operating systems provide a mechanism called *file descriptor passing*. File descriptor passing allows one process to give access to an open file to another process [25]. This is achieved by sending a control message containing the file descriptor to the other process; see Figure 1. When the message is received, the operating system creates a matching file descriptor in the file table of the receiving process that permits access to the sender's file. We implement a capability request by passing a file descriptor over the socket used for the informational requests. The capability request is an informational request in which the slave expects the monitor to answer with a control message containing the passed file descriptor.

The change of identity request is the most difficult to implement. The request is usually issued when a service changes from the pre-authentication to the post-authentication phase. After authentication, the service wants to obtain the privileges of the authenticated user. Unix operating systems provide no portable mechanism to change the user identity[1] a process is associated with unless the process has superuser privilege. However, in our case, the process that wants to change its identity does not have such privilege.

One way to effect a change of identity is to terminate the slave process and ask the monitor to create a new process that can then change its UID and GID to the desired identities. By terminating the child process all the state that has been created during its life time is lost. Normally a meaningful continuation of the session is not possible without retaining the state of the slave process. We solve this problem by exporting all state of the unprivileged child process back to the monitor.

Exporting state is messy. For global structures, we use XDR-like [16] data marshaling which allows us to package all data contained in a structure including pointers and send it to the monitor. The data is unpacked by the newly forked child process. This prevents data corruption in the exported data from affecting the privileged monitor in any way.

For structures that are allocated dynamically, *e.g.*, via *malloc*, data export is more difficult. We solve this problem by providing memory allocation from shared memory. As a result, data stored in dynamically allocated memory is also available in the address space of the privileged monitor. Figure 2 shows the interface to the shared memory allocator.

The two functions *mm_create* and *mm_share_sync* are responsible for permitting a complete export of dynamically allocated memory. The mm_create function creates a shared address space of the specified size. There are several ways to implement shared memory, we use anonymous memory maps. The returned value is a pointer to a *mm_master* structure that keeps track of allocated memory. It is used as parameter in subsequent calls to *mm_malloc* and *mm_free*. Every call to those two functions may result in allocation of additional memory for state that keeps track of free or allocated memory in the shared address space. Usually, that memory is allocated with libc's malloc function. However, the first argument to the mm_create function may be a pointer to another shared address space. In that case, the memory manager allocates space for additional state from the passed shared address space.

Figure 3 shows an overview of how allocation in the shared address space proceeds. We create two shared address spaces: *back* and *mm*. The address space *mm* uses *back* to allocate state information. When the child

[1] To our knowledge, Solaris is the only Unix operating system to provide such a mechanism.

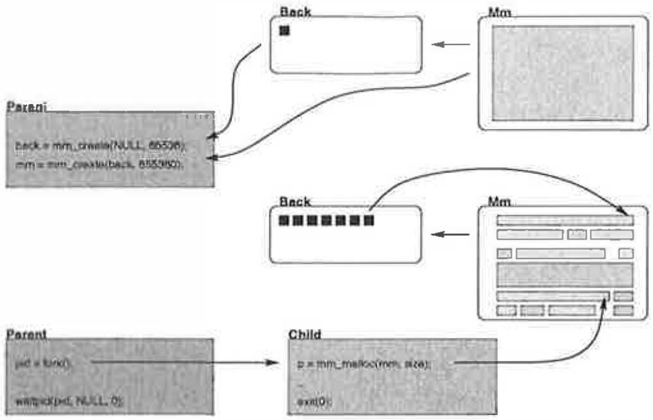


Figure 3: The complete state of a slave process includes dynamically allocated memory. When exporting this state, the dynamically allocated address space in opaque data objects must not change. By employing a shared memory allocator that is backed by another shared address space, we can export state without changing the addresses of dynamically allocated data.

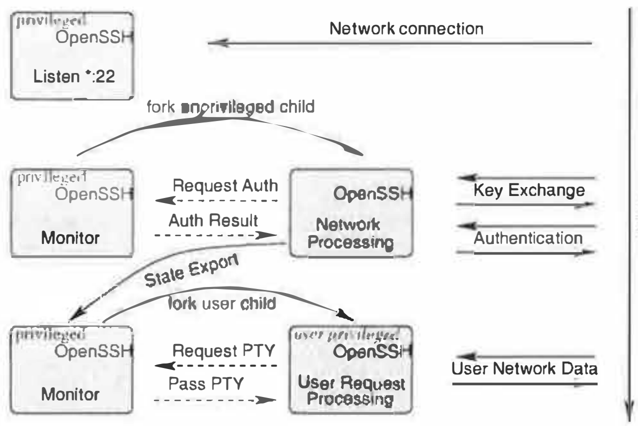


Figure 4: Overview of privilege separation in OpenSSH. An unprivileged slave processes all network communication. It must ask the monitor to perform any operation that requires privileges.

wants to change its identity, it exits and the thread of execution continues in the parent. The parent has access to all the data that was allocated in the child. However, one problem remains. The shared address space *back* uses libc's malloc that allocated memory in the child's address space to keep track of its state. If this information is lost when the child process exits, then subsequent calls to mm_malloc or mm_free fail. To solve the problem, the parent calls the mm_share_sync function which recreates the state information in the shared address space *back*. Afterwards, freeing and allocating memory proceeds without any problems.

We use shared memory and XDR-like data marshaling to export all state from the child to the parent. After the child process exports its state and terminates, the parent creates a new child process. The new process changes to the desired UID and GID and then imports the exported state. This effects a change of identity in the slave that preserves state information.

4 Separating Privileges in OpenSSH

In this section, we show how to use privilege separation in OpenSSH, a free implementation of the SSH protocols. OpenSSH provides secure remote login across the Internet. OpenSSH supports protocol versions one and two; we restrict our explanation of privilege separation to the latter. The procedure is very similar for protocol one and also applies to other services that require authentication.

When the SSH daemon starts, it binds a socket to port 22 and waits for new connections. Every new connection is handled by a forked child. The child needs to retain superuser privileges throughout its lifetime to create new pseudo terminals for the user, to authenticate key exchanges when cryptographic keys are replaced with new ones, to clean up pseudo terminals when the SSH session ends, to create a process with the privileges of the authenticated user, etc.

With privilege separation, the forked child acts as the monitor and forks a slave that drops all its privileges and starts accepting data from the established connection. The monitor now waits for requests from the slave; see Figure 4. Requests that are permitted in the pre-authentication phase are shown in Figure 5. If the child issues a request that is not permitted, the monitor terminates.

First, we identify the actions that require special privilege in OpenSSH and show which request types can fulfill them.

4.1 Pre-Authentication Phase

In this section, we describe the privileged requests for the pre-authentication phase:

- Key Exchange: SSH v2 supports the Diffie-Hellman Group Exchange which allows the client to request a group of a certain size from the server [10]. To find an appropriate group the server consults the */etc/moduli* file. However, because the slave has no privileges to access the file system, it can not open the file itself, so, it issues an informational request to the monitor. The

```
struct mon_table mon_dispatch_proto20[] = {
  {MONITOR_REQ_MODULI, MON_ONCE, mm_answer_moduli},
  {MONITOR_REQ_SIGN, MON_ONCE, mm_answer_sign},
  {MONITOR_REQ_PWNAM, MON_ONCE, mm_answer_pwnamallow},
  {MONITOR_REQ_AUTHSERV, MON_ONCE, mm_answer_authserv},
  {MONITOR_REQ_AUTHPASSWORD, MON_AUTH, mm_answer_authpassword},
[...]
  {MONITOR_REQ_KEYALLOWED, MON_ISAUTH, mm_answer_keyallowed},
  {MONITOR_REQ_KEYVERIFY, MON_AUTH, mm_answer_keyverify},
  {0, 0, NULL}
};
```

Figure 5: The table describes valid requests that a slave may send to the monitor in the pre-authentication phase for SSH protocol version two. After authentication, the set of valid requests changes and is described by a separate table.

monitor returns a suitable group after consulting the moduli file. The returned group is used by the slave for the key exchange. As seen in Figure 5, the slave may issue this request only once.

- Authenticated Key Exchange: To prevent man-in-the-middle attacks, the key exchange is authenticated. That means that the SSH client requires cryptographic proof of the server identity. At the beginning of the SSH protocol, the server sends its public key to the client for verification. As the public key is public, the slave knows it and no special request is required. However, the slave needs to ask the monitor to authenticate the key exchange by signing a cryptographic hash of all values that have been exchanged between the client and the server. The signature is obtained by an informational request.

- User Validation: After successful key exchange, all communication is encrypted and the SSH client informs the server about the identity of the user who wants to authenticate to the system. At this point, the server decides if the user name is valid and allowed to login. If it is invalid, the protocol proceeds but all authentication attempts from the client fail. The slave can not access the password database, so it must issue an informational request to the server. The server caches the user name and reports back to the slave if the name is valid.

- Password Authentication: Several methods can be used to authenticate the user. For password authentication, the SSH client needs to send a correct login and password to the server. Once again, the unprivileged slave can not access the password database, so it asks the monitor to verify the password. The monitor informs the slave if the authentication succeeds or fails. If it succeeds, the pre-authentication phase ends.

- Public Key Authentication: Public Key Authentication is similar to password authentication. If it is successful, the pre-authentication phase ends. However, two informational requests are required to use public keys for authentication. The first request allows the slave to determine if a public key presented by the client may be used for authentication. The second request determines if the signature returned by the client is valid and signs the correct data. A valid signature results in successful authentication.

At any time, the number of requests that the slave may issue are limited by the state machine. When the monitor starts, the slave may issue only the first two requests in Figure 5. After the key exchange has finished, the only valid request is for user validation. After validating the user, all authentication requests are permitted. The motivation for keeping the number of valid requests small is to reduce the attack profile available to an intruder who has compromised the slave process.

All requests up to this point have been informational. The pre-authentication phase ends with successful authentication as determined by the monitor. At this point, the slave needs to change its identity to that of the authenticated user. As a result, the slave obtains all privileges of the user, but no other privileges. We achieve this with a change of identity request.

The monitor receives the state of the slave process and waits for it to exit. The state consists of the following: the encryption and authentication algorithms including their secret keys, sequence counters for incoming and outgoing packets, buffered network data and the compression state.

Exporting the cryptographic key material is uncomplicated. The main problem is exporting the compression state. The SSH protocols use the *zlib* compression format [7, 8] which treats network data as a stream instead of sequence of packets. Treating network data as a stream allows zlib to improve its dictionary with increasing amount of compressed data. On the other hand, it also means that compression in the server cannot be stopped and then restarted as the client uses a dictionary that depends on all the preceding data. Fortunately, zlib provides hooks for user supplied memory management functions. We provide it with functions that use *mm_malloc* and *mm_free* as back end. After the child exits, the monitor needs only to call *mm_share_sync* to import the compression state.

4.2 Post-Authentication Phase

The monitor forks a new process that then changes its process identification to that of the authenticated user. The slave process obtains all the privileges of the authenticated user. At this point, we enter the post-authentication phase which requires only a few privileged operations. They are as follows:

- Key Exchange: In SSH protocol version two, it is possible to renew cryptographic keys. This requires a new key exchange, so just as in the pre-authentication phase, the monitor chooses a suitable group for the Diffie-Hellman key exchange and signs for authentication.
- Pseudo Terminal Creation: After authentication, the user requires a pseudo terminal whose creation requires superuser privileges. For a Unix application, a pseudo terminal is just a file descriptor. The slave issues a capability request to the monitor. The monitor creates the terminal and passes the corresponding file descriptor to the child process. An informational request suffices when the slave wants to close the pseudo terminal.

4.3 Discussion

Observe that the majority of all privileged operations can be implemented with informational requests. In fact, some degree of privilege separation is possible if neither capability nor change of identity requests are available. If the operating system does not support file descriptor passing, privilege separation perforce ends after the pre-authentication phase. To fully support the change of identify request shared memory is required. Without shared memory, the compression state cannot be exported without rewriting *zlib*. Nonetheless, systems that do not support shared memory can disable compression and still benefit from privilege separation.

Using an alternative design, we can avoid the change of identity request and shared memory. Instead of using only two processes: monitor and slave, we use three processes: one monitor process and two slave processes. The first slave operates similarly to the slave process described in the pre-authentication phase. However, after the user authenticates, the slave continues to run and is responsible for encrypting and decrypting network traffic. The monitor then creates a second slave to execute a shell or remote command with the credentials of the authenticated user. All communication passes via the first child process to the second. This design requires no state export and no shared memory. Although the cryptographic processing is isolated in the first child, it has only a small effect on security. In the original design, a bug in the cryptographic processing may allow an adversary to execute commands with the privilege of the authenticated user. However, after authentication, an adversary can already execute any commands as that user. The three process design may help for environments in which OpenSSH restricts the commands a user is allowed to execute. On the other hand, it adds an additional process, so that every remote login requires three instead of two processes. While removing the state export reduces the complexity of the system, synchronizing three instead of two processes increases it. An additional disadvantage is a decrease in performance because the three process design adds additional data copies and context switches.

For the two process design, the changes to the existing OpenSSH sources are small. About 950 lines of the 44,000 existing lines of source code, or 2%, were changed. Many of the changes are minimal:

```
- authok = auth_password(authctxt, pwd);
+ authok = PRIVSEP(auth_password(authctxt, pwd);
```

The new code that implements the monitor and the data marshaling amounts to about three thousand lines of source code, or about seven percent increase in the size of the existing sources.

While support for privilege separation increases the source code size, it actually reduces the complexity of the existing code. Privilege separation requires clean and well abstracted subsystem interfaces so that their privileged sections can run in a different process context. During the OpenSSH implementation, the interfaces for several subsystems had to be improved to facilitate their separation. As a result, the source code is better organized, more easily understood and audited, and less complex.

The basic functionality that the monitor provides is independent of OpenSSH. It may be used to enable

privilege separation in other applications. We benefit from reusing security critical source code because it results in more intense security auditing. This idea has been realized in Privman, a library that provides a generic framework for privilege separation [12].

5 Security Analysis

To measure the effectiveness of privilege separation in OpenSSH, we discuss attacks that we protect against and analyse how privilege separation would have affected security problems reported in the past. We assume that the employed cryptography is secure, therefore we do not discuss problems of cryptographic primitives.

After privilege separation, two thirds of the source code are executed without privileges as shown in Table 1. The numbers include code from third-party libraries such as *openssl* and *zlib*. For OpenSSH itself, only twenty five percent of the source code require privilege whereas the remaining seventy five percent are executed without special privilege. If we assume that programming errors are distributed fairly uniformly, we can estimate the increase of security by counting the number of source code lines that are now executed without privileges. This back of the envelope analysis suggests that two thirds of newly discovered or introduced programming errors will not result in privilege escalation and that only one third of the source code requires intensive auditing.

We assume that an adversary can exploit a programming error in the slave process to gain complete control over it. Once the adversary compromised the slave process, she can make any system call in the process context of the slave. We assume also that the system call interface to the operating system itself is secure[2]. Still, there are several potential problems that an implementation of privilege separation needs to address:

- The adversary may attempt to signal or ptrace other processes to get further access to the system. This is not possible in our design because the slave processes use their own UID.

- The adversary may attempt to signal or ptrace the slave processes of other SSH sessions. When changing the UID of a process from root to another UID, the operating system marks the process as `P_SUGID` so that only root may signal or ptrace it. As a result, a slave process can not signal another slave.

- She may attempt to use system calls that change the file system, for example to create named pipes for interprocess communication or device nodes. However, as a non-root user the slave process has its file system root set to an empty read-only directory that the adversary can not escape from.

- Using privilege separation, we cannot prevent the adversary from initiating local network connections and potentially abusing trust relations based on IP addresses. However, we may restrict the child's ability to access the system by employing external policy enforcement mechanisms like Systrace [21].

- The adversary may attempt to gather information about the system, for example, the system time or PIDs of running processes, that may allow her to compromise a different service. Depending on the operating system, some information is exported only via the file system and can not be accessed by the adversary. A sandbox may help to further restrict the access to system information.

Another way an adversary may try to gain additional privileges is to attack the interface between the

[2]This assumption does not always hold. A bug in OpenBSD's select system call allowed an adversary to execute arbitrary code at the kernel-level [5, 20].

Subsystem	Lines of Code	Percentage
Unprivileged	**17589**	**67.70%**
OpenSSH	*10360*	*39.88%*
Ciphers	267	1.03%
Packet Handling	1093	4.21%
Miscellaneous	7944	30.58%
Privsep Interface	1056	4.06%
OpenSSL	*3138*	*12.08%*
Diffie Hellman	369	1.42%
Symmetric Ciphers	2769	10.66%
Zlib	*4091*	*15.75%*
Privileged	**8391**	**32.30%**
OpenSSH	*3403*	*13.10%*
Authentication	803	3.09%
Miscellaneous	1700	6.54%
Monitor	900	3.46%
OpenSSL	*4109*	*15.82%*
BigNum/Hash	3178	12.23%
Public Key	931	3.58%
SKey	*879*	*3.17%*

Table 1: Number of source code lines that are executed with and without privileges.

privileged monitor and the slave. The adversary could send badly formatted requests in the hope of exploiting programming errors in the monitor. For that reason, it is important to carefully audit the interface to the monitor. In the current implementation, the monitor imposes strict checks on all requests. Furthermore, the number of valid requests is small and any request detected as invalid causes the privileged monitor to terminate.

Nonetheless, there may be other ways that an adversary might try to harm the system. She might try to starve the resources of the system by forking new processes or by running very time intensive computations. As a result, the system might become unusable. The effect of such an attack can be mitigated by placing process limits on the slave process. For example, we can limit the number of file descriptors the slave may open and the number of processes it is allowed to fork. The monitor may also watch other resource utilization like CPU time and terminate the slave if a certain threshold is reached.

In the following, we discuss how privilege separation would have affected previous progamming errors in OpenSSH.

The *SSH-1 Daemon CRC32 Compensation Attack Detector Vulnerability* permits an adversary to gain superuser privileges remotely without authenticating to the systems [31]. The problem is caused by an integer overflow in a function that processes network packets. With privilege separation, this function is executed without any privileges, which makes it impossible for an adversary to directly compromise the system.

Similarly, the off-by-one error in OpenSSH's channel code allows an adversary to gain superuser privileges after authenticating to the system [19]. With privilege separation, the process has only the privileges of the authenticated user. An adversary cannot obtain system privileges in this case either.

A security problem in the external *zlib* compression library was found that might allow a remote adversary to gain superuser privileges without any authentication [3]. This problem occurs in a third-party library, so no audit of the OpenSSH source code itself can find it. Privilege separation prevents a system compromise in this case, too.

At the time of this writing, additional security problems have been found in OpenSSH. A bug in the Kerberos ticket passing functions allowed an authenticated user to gain superuser rights. A more severe problem in code for challenge-response authentication allows a remote adversary to obtain superuser privileges without any authentication [4]. Privilege separation prevents both of these problems and is mentioned in the CERT advisory as a solution.

The programming errors in the channel code and in the Kerberos ticket passing functions occur in the post-authentication phase. Without privilege separation, these errors allow an authenticated user to gain superuser privilege. The remaining errors occur during pre-authentication and may allow an adversary to gain superuser privilege without any authentication if privilege separation is not used.

These examples demonstrate that privilege separation has the potential to contain security problems yet unknown.

6 Performance Analysis

To analyze the performance of privilege separated OpenSSH, we measure the execution time for several different operations in monolithic OpenSSH and the privileged separated version. We conduct the measurements on a 1.13 GHz Pentium III laptop with all data in the memory cache.

Test	Normal	Privsep
Login		
- compressed	0.775s $\pm$ 0.0071s	0.777s $\pm$ 0.0067s
- uncompressed	0.767s $\pm$ 0.0106s	0.774s $\pm$ 0.0097s
Data Transfer		
- compressed	4.229s $\pm$ 0.0373s	4.243s $\pm$ 0.0411s
- uncompressed	1.989s $\pm$ 0.0223s	1.994s $\pm$ 0.0143s

Table 2: Performance comparison between normal OpenSSH and privilege separated OpenSSH. We measure the overhead in login and data transfer time when employing privilege separation. In both cases, privilege separation imposes no significant performance penality.

The first test measures the time it takes to login using public key authentication. We measure the time with compression enabled and without compression. The next two tests measure the data transfer time of a 10 MB file filled with random data, with compression enabled, and without compression. The results are shown in Table 2. It is evident that privilege separated OpenSSH does not penalize performance. As the IPC between monitor and slave is never used for moving large amounts of data, this is not surprising.

7 Related Work

Confidence in the security of an application starts by source code inspection and auditing. Static analysis

provides methods to automatically analyze a program's source code for security weaknesses. Using static analysis, it is possible to automatically find buffer overrun vulnerabilities [13, 27], format string vulnerabilities [24], etc.

While source code analysis enables us to find some security vulnerabilities, it is even more important to design applications with security in mind. The principle of least privilege is a guideline for developers to secure applications. It states that every program and every user should operate using the least amount of privilege necessary to complete the job [22].

Security mechanisms at the operating system level provide ways to reduce the privileges that applications run with [1, 29, 18, 21]. However, these mechanisms are unaware of an application's internal state. For example, they cannot determine if users authenticate successfully. As a result, they have to allow all operations of authenticated users even when attached by an adversary. Privilege separation remedies this problem because it is built into the application and exposes only an unprivileged child to the adversary. There are several applications that make use of privilege separation as we discuss below. The main difference in this research is the degree and completeness of the separation.

Carson demonstrates how to reduce the number of privileges that are required in the *Sendmail* mail system [2]. His design follows the principle of least privilege. While Sendmail is a good example, the degrees of privilege separation demonstrated in OpenSSH are much more extensive. For example, we show how to change the effective UID and how to retain privileges securely for the whole duration of the session.

Venema uses semi-resident, mutually-cooperating processes in Postfix [26]. He uses the process context as a protection domain similar to our research in privilege separation. However, a mail delivery system does not require the close interaction between privileged and unprivileged processes as necessary for authentication services like OpenSSH. For system services that require transitions between different privileges, our approach seems more suitable.

Evans *very secure* FTP daemon uses privilege separation to limit the effect of programming errors [9]. He uses informational and capability requests in his implementation. His work is very similar to the implementation of privilege separation in OpenSSH, but not as extensive and less generic.

Solar Designer uses a process approach to switch privileges in his *Owl* Linux distribution [6]. His POP3 daemon *popa3d* forks processes that execute protocol operations with lower privileges and communicate results back to the parent. The interaction between parent and child is based completely on informational requests.

Separating the privileges of an application causes a decomposition into subsystems with well defined functionality. This is similar to the design and functionality of a microkernel where subsystems have to follow the principle of independence and integrity [14]. For a privilege separated application, independence and integrity are realized by multiple processes that have separate address spaces and communicate via IPC.

8 Conclusion

Programming errors in privileged services can result in system compromise allowing an adversary to gain unauthorized privileges.

Privilege separation is a concept that allows parts of an application to run without any privileges at all. Programming errors in the unprivileged part of the application cannot lead to privilege escalation.

As a proof of concept, we implemented privilege separation in OpenSSH and show that past errors that allowed system compromise would have been contained with privilege separation.

There is no performance penalty when running OpenSSH with privilege separation enabled.

9 Acknowledgments

We thank Solar Designer, Dug Song, David Wagner and the anonymous reviewers for helpful comments and suggestions.

References

[1] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, and Sheila A. Haghighat. A Domain and Type Enforcement UNIX Prototype. In *Proceedings of the 5th USENIX Security Symposium*, pages 127–140, June 1995. 10

[2] Mark E. Carson. Sendmail without the Superuser. In *Proceedings of the 4th USENIX Security Symposium*, October 1993. 10

[3] CERT/CC. CERT Advisory CA-2002-07 Double Free Bug in zlib Compression Library. `http://www.cert.org/advisories/CA-2002-07.html`, March 2002. 9

[4] CERT/CC. CERT Advisory CA-2002-18 OpenSSH Vulnerabilities in Challenge Response Handling. `http://www.cert.org/advisories/CA-2002-18.html`, June 2002. 9

[5] Silvio Cesare. FreeBSD Security Advisory FreeBSD-SA-02:38.signed-error. `http://archives.neohapsis.com/archives/freebsd/2002-08/0094.html`, August 2002. 8

[6] Solar Designer. Design Goals for popa3d. `http://www.openwall.com/popa3d/DESIGN`. 10

[7] P. Deutsch. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951, 1996. 7

[8] P. Deutsch and J-L. Gailly. ZLIB Compressed Data Format Specification version 3.3. RFC 1950, 1996. 7

[9] Chris Evans. Comments on the Overall Architecture of Vsftpd, from a Security Standpoint. `http://vsftpd.beasts.org/`, February 2001. 10

[10] Markus Friedl, Niels Provos, and William A. Simpson. Diffie-Hellman Group Exchange for the SSH Transport Layer Protocol. Internet Draft, January 2002. Work in progress. 5

[11] Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2. *USENIX Symposium on Internet Technologies and Systems*, pages 103–112, 1997. 1

[12] Douglas Kilpatrick. Privman: A Library for Partitioning Applications. In *Proceedings of the USENIX 2003 Annual Technical Conference, FREENIX track*, June 2003. 8

[13] David Larochelle and David Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, August 2001. 10

[14] Jochen Liedtke. On μ-Kernel Construction. In *Proceedings of the Symposium on Operating Systems Principles*, pages 237–250, 1995. 10

[15] Theodore A. Linden. Operating System Structures to Support Security and Reliable Software. *ACM Computing Surveys*, 8(4):409–445, 1976. 2

[16] Sun Microsystems. XDR: External Data Representation. RFC 1014, June 1987. 4

[17] Todd C. Miller and Theo de Raadt. strlcpy and strlcat – Consistent, Safe, String Copy and Concatenation. In *Proceedings of the 1999 USENIX Technical Conference, FREENIX track*, June 1999. 2

[18] David S. Peterson, Matt Bishop, and Raju Pandey. A Flexible Containment Mechanism for Executing Untrusted Code. In *Proceedings of the 11th USENIX Security Symposium*, pages 207–225, August 2002. 1, 10

[19] Joost Pol. OpenSSH Channel Code Off-By-One Vulnerability. `http://online.securityfocus.com/bid/4241`, March 2002. 9

[20] Niels Provos. OpenBSD Security Advisory: Select Boundary Condition. `http://monkey.org/openbsd/archive/misc/0208/msg00482.html`, August 2002. 8

[21] Niels Provos. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium*, August 2003. 1, 8, 10

[22] Jerome H. Saltzer. Protection and the Control of Information in Multics. *Communications of the ACM*, 17(7):388–402, July 1974. 10

[23] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. In *Proceedings of the IEEE 69*, number 9, pages 1278–1308, September 1975. 2

[24] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, August 2001. 10

[25] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1992. 4

[26] Wietse Venema. Postfix Overview. `http://www.postfix.org/motivation.html`. 10

[27] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proceedings of the ISOC Symposium on Network and Distributed System Security*, pages 3–17, February 2000. 10

[28] David A. Wagner. Janus: an Approach for Confinement of Untrusted Applications. Technical Report CSD-99-1056, University of California, Berkeley, 12, 1999. 1

[29] Kenneth M. Walker, Daniel F. Sterne, M. Lee Badger, Michael J. Petkac, David L. Shermann, and Karen A. Oostendorp. Confining Root Programs with Domain and Type Enforcement (DTE). In *Proceedings of the 6th Usenix Security Symposium*, July 1996. 10

[30] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible Security Architectures for Java. *16h Symposium on Operating System Principles*, pages 116–128, 1997. 1

[31] Michal Zalewski. Remote Vulnerability in SSH Daemon CRC32 Compensation Attack Detector. `http://razor.bindview.com/publish/advisories/adv_ssh1crc.html`, February 2001. 9

Dynamic Detection and Prevention of Race Conditions in File Accesses

Eugene Tsyrklevich
eugene@securityarchitects.com

Bennet Yee
bsy@cs.ucsd.edu

Department of Computer Science and Engineering
University of California, San Diego

Abstract

Race conditions in filesystem accesses occur when sequences of filesystem operations are not carried out in an isolated manner. Incorrect assumptions of filesystem namespace access isolation allow attackers to elevate their privileges without authorization by changing the namespace bindings. To address this security issue, we propose a mechanism for keeping track of all filesystem operations and possible interferences that might arise. If a filesystem operation is found to be interfering with another operation, it is temporarily suspended allowing the first process to access a file object to proceed, thereby reducing the size of the time window when a race condition exists. The above mechanism is shown to be effective at stopping all realistic filesystem race condition attacks known to us with minimal performance overhead.

1 Introduction

Dating back to the 1970's, race conditions, otherwise known as the Time Of Check To Time Of Use (TOCTTOU) bug, are one of the oldest known security flaws. They are a class of local vulnerabilities that can be used to elevate one's privileges without authorization and occur when operations are not carried out atomically as expected.

Race conditions have been found in a variety of scenarios ranging from tracing processes [9] to filesystem accesses [13]. This paper focuses on a subset of race conditions dealing with filesystem accesses. The threat model is that of an adversary with unprivileged access to a system who is trying to elevate his or her privileges by exploiting a filesystem race condition vulnerability.

Some security vulnerabilities, such as buffer overflows, receive a lot of attention and have a variety of solutions to prevent them. Other vulnerabilities, such as race conditions, receive far less attention and have no definite solutions to stop them. In addition, race conditions can be very hard to spot and eliminate at compile time, it is thus beneficial to develop a dynamic and automated protection against this type of flaws.

This paper presents an analysis of filesystem race condition vulnerabilities and ways to prevent them. It also describes a prototype system that detects and stops all realistic filesystem race condition attacks known to us. The proposed solution stops the attacks with minimum performance overhead, while retaining application compatibility.

The rest of this paper is organized as follows. Section 2 describes race conditions in filesystem accesses and presents several sample vulnerabilities. The design of the prototype system and the rationale behind it is covered in Section 3. Section 4 details the implementation of the prototype system, as well as some of the decisions made during the design phase incorporating the security of the system and code portability. Section 5 reviews the attack scenarios used for the security testing of the prototype system. Section 6 describes the compatibility testing and Section 7 covers the system performance. Section 8 reviews related work, Section 9 proposes future work and Section 10 provides the conclusions.

2 Race Conditions in File Accesses

Race conditions in filesystem accesses occur when applications incorrectly assume that a sequence of filesys-

```
if (access(fn, R_OK | W_OK) == 0) {
      fd = open(fn, O_RDWR, 0644);
      if (fd < 0)
      {
            err(1, fn);
      }
      /* use the file */
```

Figure 1: Race Condition in Access Operation

tem operations is atomic and the filesystem namespace is otherwise static. In reality, attackers can modify the filesystem namespace at any time by replacing files with symbolic links or by renaming files and directories.

2.1 Access Operation

The code fragment in Figure 1 is taken from an insecure setuid application that needs to check whether a user running the program has the right to write to a file (since the application is setuid root, it can write to any file).

The source code incorrectly assumes that a file object referenced by the filename "fn" does not change in between the `access` and `open` invocations. The assumption is invalid since an attacker can replace the file object "fn" with another file object (such as a symbolic link to another file), tricking the program into accessing arbitrary files. To avoid this race condition, an application should change its effective id to that of a desired user and then make the `open` system call directly or use `fstat` after the `open` instead of invoking `access`.

2.2 Directory Operations

Another subtle race condition was recently discovered in the GNU file utilities package [13]. During the recursive removal of a directory tree, an insecure `chdir("..")` operation is performed, after removing the content of a subdirectory, to get back to the upper directory. An attacker, who is able to exploit this race condition, can trick users into deleting arbitrary files and directories. The bug is triggered when an attacker moves a subdirectory, that the rm utility is processing, to a different directory.

Figure 2 contains a slightly abbreviated system call trace of a recursive removal of the `/tmp/a` directory tree that contains two nested subdirectories. Consider what happens when an attacker moves `/tmp/a/b/c` directory to `/tmp` before the first `chdir("..")` call is invoked. After removing the contents of the `/tmp/a/b/c` subdirectory (now `/tmp/c`) and executing two `chdir("..")` calls, instead of changing the directory to `/tmp/a`, the `rm` utility would find itself in the root directory that it would continue to delete.

```
chdir("/tmp/a")
chdir("b")
chdir("c")
unlink("*")
[attacker entrypoint: "mv /tmp/a/b/c /tmp"]
chdir("..")
rmdir("c")
unlink("*")
chdir("..")
rmdir("b")
unlink("*")
rmdir("/tmp/a")
```

Figure 2: Race Condition in Directory Operations

The race condition was fixed by adding additional checks to verify that the inode of the directories did not change before and after the `chdir` calls; in other words, the directory tree namespace did not change during the execution of the tool.

2.3 Setuid Shell Scripts

The secure execution of setuid shell scripts is a well-known problem in the earlier Unix systems. To determine whether a file is a script and should be processed by an interpreter, a Unix kernel checks the first two bytes of a file for a special "`#!`" character sequence. If the character sequence is found, the rest of the line is treated as the interpreter pathname (and its parameters) that should be executed to process the script. Unfortunately, in some kernels, the operation of opening and processing the first line of a file and the subsequent execution of the interpreter that reopens the script is not atomic. This allows attackers to trick the operating system into processing one file while executing the interpreter with root privileges on another file.

Various Unix systems deal with this problem in differ-

```
if (stat(fn, &sb) == 0) {
    fd = open(fn, O_CREAT | O_RDWR, 0);
    if (fd < 0)
    {
        err(1, fn);
    }

    /* use the file */
```

Figure 3: Race Condition in Temporary File Accesses

ent ways. Earlier Linux systems simply disregarded the setuid bits when executing the shell scripts. Some SysV systems as well as BSD 4.4 kernels pass to the interpreter a filename in a private, per-process filesystem namespace representing the already opened descriptor to the script (e.g. `/dev/fd/3`) in lieu of the original pathname. Passing such a pathname closes down the race condition vulnerability window since it is impossible for an attacker to change the private per-process filesystem namespace bindings.

2.4 Temporary File Accesses

Another type of race condition occurs in the handling of temporary files. An unsafe way to create a temporary file is to use the `stat` system call to check whether a file exists and if the file does not exist, invoke `open` to create one. The problem with this approach is that the sequence of `stat` followed by `open` is not atomic (similarly to the `access` and `open` system call sequence described above). This allows an attacker to create a symbolic link with the same filename after the `stat` reports that the file does not exist. When the `open` system call with the `O_CREAT` flag executes, it follows the attacker's link and overwrites the contents of the file to which the link points. Figure 3 shows the vulnerable code.

The correct way to do the same thing would be to invoke `open` with `O_CREAT` and `O_EXCL` flags set. The `O_EXCL` flag causes the `open` to fail when the file already exists.

3 System Design

The examples in the previous section illustrated that race conditions could occur under a wide variety of conditions. The theme that is present in all of the examples is that programmers erroneously assumed that their program is the only program accessing the filesystem. Instead, an attacker can change the filesystem namespace in the midst of a sequence of filesystem system calls and exploit the race condition in the code. We call the sequence of filesystem system calls that programmers assumed to be free from interference a *pseudo-transaction*. To address the problem of race conditions, it would be sufficient to provide the illusion of a filesystem namespace that behaves as if these pseudo-transactions were executed in isolation and free from interference. To achieve this, the system needs to be able to detect race condition attacks and stop them.

3.1 Transactions

The goal of detecting and preventing race conditions would be trivially achieved if executions of processes (or process groups) were transactions that enjoyed the ACID properties (Atomicity, Consistency, Isolation and Durability). ACID transactions were designed to permit concurrent database accesses while maintaining correctness. These ideas can be readily translated to filesystem accesses.

The atomicity property requires that a transaction commits all the involved data or none at all.

The consistency property guarantees that a transaction produces only consistent results; otherwise it aborts. Therefore, filesystem operations belonging to a transaction always produce coherent results without external interferences.

The isolation property requires that a running transaction remains isolated from all other running transactions. In the race protection system, this means that the filesystem operations in one transaction cannot effect the results of filesystem operations belonging to other transactions. This is the main characteristic that the race protection system tries to achieve.

Finally, durability requires that the results of the completed transactions must not be forgotten by the system.

The ideal operating system for the task would satisfy all of the above properties. QuickSilver is an example of one such system that provides built-in transaction primitives [14]. Unfortunately, the mainstream operating systems have no built-in support for transaction processing. In the absence of transaction processing, the race protection system has to at least guarantee the isolation property which requires the system to be able to detect the

race condition attacks before they take place. Furthermore, the system must be able to enforce the property by preventing the race conditions from being exploited.

3.2 Detecting Race Conditions

Unfortunately, application programmers do not include code that informs the kernel when a pseudo-transaction begins or ends. We can, however, use heuristics to analyse filesystem-related activities and attempt to identify the pseudo-transactions. Below, we address how we identify the start of pseudo-transactions and policies to determine if a filesystem operation request would interfere with an in-progress pseudo-transaction.

3.2.1 Scope of Pseudo-Transactions

By monitoring filesystem operations, the system tries to distinguish between normal and potentially malicious filesystem activities. First, the simple heuristics used for this recognize system calls that potentially start pseudo-transactions. Then, they prevent subsequent system calls from interfering with the pseudo-transaction. Filesystem calls that do not interfere are executed normally, while those that do are handled as described in Section 3.3. In our analysis of filesystem activities, we found that pseudo-transactions that involve sequences of more than two operations are not common. As a result, they are not addressed by our implementation. This allows us to end a potential pseudo-transaction at the next allowed system call, which may in turn start a new pseudo-transaction. We also allow a pseudo-transaction to end via a heuristically determined time out value (see Section 4.3); this ensures that misidentified pseudo-transactions will not persist in the system for too long.

We assume that a process does not maliciously interfere with its own filesystem accesses. Furthermore, `fork`'ed child processes run the same process image and are assumed to be trustworthy until an `exec` replaces their process image. This means that the granularity of detection is at the process level—all filesystem operations originating from a process will be treated as part of a single pseudo-transaction, even if the process performs work on behalf of multiple clients (e.g., a web server).

```
REMOVE = UNLINK | RMDIR | RENAME

DENY(ACCESS, REMOVE)
DENY(CHDIR, REMOVE)
DENY(EXEC, REMOVE)
```

Figure 4: Default Allow Policy

3.2.2 Policies

In order to detect potential race conditions, we propose two different policies. One policy disallows operation sequences that are prone to filesystem races, while the second policy only allows the operation sequences that are known to be safe. Both policies take into consideration the current filesystem operation and the last operation that dealt with the same file. If the operation sequence is found to be legal by the chosen policy, the current operation is allowed to proceed. Otherwise, the policy returns an error that indicates that a potential race condition is in progress.

Default Allow Policy As the name suggests, a default allow policy permits all filesystem operations to proceed unless they match one of the deny rules. Deny rules (listed in Figure 4) are constructed to address and stop specific race condition attacks. For example, the `DENY(ACCESS,REMOVE)` rule does not allow the `access` operation to be followed by any of the remove operations if they involve the same file object. All other filesystem operations, however, are allowed to proceed.

The above mentioned rule prevents the attack described in Section 2.1, the second deny rule addresses the race condition described in Section 2.2 while the third rule deals with the attack described in Section 2.3. Race conditions in temporary file accesses are treated separately as described in Section 4.2.

A default allow policy explicitly forbids sequences of filesystem operations that are prone to race conditions. As a result, the policy should detect all listed race condition attacks without causing any false positives. However, it may miss new attacks if they involve operation sequences not explicitly stated in the policy.

Default Deny Policy A default deny policy disallows all filesystem operations to proceed unless they match one of the permit rules. Permit rules (listed in Figure 5) are constructed to describe all known

```
OPEN_RW = OPEN_READ | OPEN_WRITE
RENAME = RENAME_TO | RENAME_FROM

PERMIT(OPEN_RW,            OPEN_RW | ACCESS | UTIMES | CHDIR | EXEC |
                           UNLINK | READLINK | CHMOD | CHOWN | RENAME)
PERMIT(OPEN_CREAT,         OPEN_RW | ACCESS | UTIMES | CHDIR | EXEC | RENAME_FROM)
PERMIT(ACCESS,             OPEN_RW | ACCESS | UTIMES | CHDIR |EXEC)
PERMIT(EXEC,               OPEN_READ | EXEC)
PERMIT(CHDIR,              OPEN_READ | CHDIR | ACCESS | READLINK)
PERMIT(RENAME_FROM,        OPEN_RW | ACCESS | UNLINK | RENAME_FROM)
PERMIT(RENAME_TO,          OPEN_RW)
PERMIT(CHMOD | CHOWN,      OPEN_RW | ACCESS |  CHMOD | CHOWN)
PERMIT(UTIMES,             OPEN_RW | ACCESS | CHMOD | CHOWN)
PERMIT(READLINK,           READLINK)
```

Figure 5: Default Deny Policy

valid filesystem operation sequences that we have observed in a normal system. For example, the `PERMIT(EXEC,OPEN_READ|EXEC)` rule allows an `exec` operation to be followed by another `exec` or by an `open` operation. All other operations involving the same file object will be flagged as race conditions.

The effectiveness and robustness of the race protection system depends on the policy in place and the quality of the rules in the chosen policy. A default deny approach should prevent any existing race condition attacks and possibly some of the new future attacks.

Future attacks are addressed by the existing rules that deal with all filesystem operations and not just the ones that are known to be susceptible to race conditions. Since each rule is a simple C macro that expands to an if statement, even a large number of rules should have negligible effect on the overall system performance.

3.3 Preventing Race Conditions

Once a race condition is identified, an error message is logged and a preventative measure must be taken. An error message should contain enough information to track down who caused the race condition and how. The approach implemented in our system is described in Section 3.3.5.

3.3.1 Transaction Rollback

One of the most sophisticated way to handle a race condition is to restart or abort the involved transactions (provided a transaction-enabled system such as QuickSilver [14] is used). Aborting the transactions should rollout any possible changes that the processes made up to that point. This solution stops race conditions without leaving the system in an inconsistent state.

3.3.2 User Confirmation

In the absence of a transaction-enabled system, the system might ask a user what to do. Similar to a systrace GUI [12], a pop-up dialog might query a user about a possible action to be taken. This approach has the advantage of not being intrusive as it is up to the user to decide whether to kill a process, suspend a process or take some other action. However, interacting with users is more suitable for desktop machines rather than servers which might not have live personnel in front of them 24/7. In addition, an attacker might overwhelm a system with a high number of alerts causing a user to ignore the numerous dialog boxes.

3.3.3 Locking Out Processes

Another option that, at first glance, would seem to work well is to preclude any other processes from being scheduled during a sequence of critical (race-prone) accesses. Programs can be "wrapped" with a script which

tells the scheduler to protect them from interfering accesses. This approach is analogous to having the process place a single lock on the entire filesystem.

While this would work in theory, there are many drawbacks that make this approach impractical. It is not clear whether it is possible to determine all the vulnerable critical sections correctly. As a data point, after kernel support for setuid script execution was added to the BSD kernel, about a decade passed before the race condition was noticed. It is also not clear whether this approach might cause deadlocks or invite denial-of-service attacks. Finally, since this locking system is very coarse grained, the system performance would suffer greatly.

3.3.4 Killing Processes

Another way way to prevent a race condition is to kill the involved processes. An alternative might be to silently fail all the consecutive system calls belonging to the involved processes and hope that the applications can cope with this and gracefully shutdown. While these solutions definitely prevent any possible abuse, they are too crude and are subject to denial-of-service attacks where an attacker might try to trick the system into killing "innocent" victim processes.

3.3.5 Suspending Processes

A better solution would minimize the effect of false positives—without killing the involved processes or otherwise completely locking files and preventing forward progress—and still prevent filesystem race attacks. Our system achieves this by allowing only one process to go on while *temporarily* suspending the other. In an attack scenario, this approach allows the victim process (the first to access a file object) to safely continue using the file while suspending the activity of an attacker process. Since a race condition is defined to exist only for brief amounts of time, delaying an attacker process is an effective way to prevent race conditions. In the scenario where two processes just happened to execute an unsafe combination of filesystem operations, the proposed solution of suspending a process allows one process to continue while preventing the second process from causing any unintentional damage. As both processes are eventually allowed to proceed, this scenario is equivalent to that of a highly-loaded machine where processes can be suspended or swapped out for extended periods of time before being allowed to run.

When a process invokes a system call that initiates a pseudo-transaction, it is marked as a "delay-lock owner" for the file. Processes attempting to access delay-locked files with interfering system calls are not completely locked out, but are instead temporarily suspended to try to prevent interference with the pseudo-transaction filesystem sequence. Delay-lock owners of files are never misidentified as interfering processes and thus delayed. Delay-lock ownership is inherited across `fork`'s and is given up at `exec`'s. Therefore, a delay lock may have many owners.

Note that we do not distinguish between parent and child processes in determining which is the attacker and which is the victim. Pseudo-transaction recognition is based solely on the global stream of system calls. Although the `fork` system call (see Section 4.4) is mediated, we only use it to allow child processes to inherit delay-lock ownership. Because we do not use per-process filename caches nor the asymmetric cache clearing scheme used by RaceGuard [6], an attack due to Casper Dik [7] is impossible.

Suspending a process is an effective technique to prevent race conditions, but it is not completely foolproof. If the attacker process wakes up before the victim process gets a chance to complete its operations, the race condition will still be present. To address this issue, the delay is calculated at runtime by taking into account the load average of a system. It would also be possible to allow a user application to control the delay by providing a new system call, but we did not implement this. While we have chosen the delay to be long enough for a process to be able to "close" its vulnerability time window, the system does not in any way detect or enforce that this actually holds.

4 Implementation

The prototype system was developed for the OpenBSD operating system [10], which was chosen for its focus on security. The system consists of a kernel module that intercepts a number of system calls. Even though modifying the operating system kernel code directly would provide a slightly faster solution, using kernel modules saves time writing and debugging the kernel code. It should be noted that mediating system calls introduces a race condition whereby an attacker changes the filename after it is processed by the mediated call but before it is processed by the original system call [8]; an integrated implementation that implements our technique directly

in the kernel code would not leave this (small) timing window open.

4.1 File System Calls Mediation

The system calls that the kernel module intercepts are mainly the filesystem calls that accept pathnames as their parameters. The mediated system calls perform processing on the pathnames before returning the control to the original system calls. The pathname processing starts with the conversion of a name to its inode by means of the `namei` kernel function. The reason for dealing with inodes rather than names is to properly handle different pathnames that refer to the same file (e.g. `/etc/passwd` vs `/etc/../etc/passwd`).

Once a pathname is resolved to an inode, the system checks if any of the preceding operations affected the same inode. If this is the case, further checks need to be carried out to make sure no race conditions exist. Otherwise, all the related operation information is saved and the operation is allowed to proceed.

As mentioned above, there are two standpoints that can be adopted to determine whether a race condition exists between the two filesystem operations involving the same inode: default allow and default deny policies. With either policy in place, the system first compares the process ids of the involved processes. If they are the same, a race condition cannot exist since both operations originated from the same process. If not, the two operations are checked against the chosen ruleset. If the operation sequence is found to be legal, the information about the current operation is saved for later use. Otherwise, an error code is returned and the appropriate action to prevent the race condition is carried out (e.g., a process is temporarily suspended). It should be noted that, in the case of a false positive, the applications continue to make forward progress, albeit at a slower rate.

4.2 Temporary File Race Condition Processing

Temporary file race condition processing differs from the above described procedure. When the `stat` system call is invoked on a non-existent file, there is no inode value to process (since the file does not exist) and therefore the inode processing code fails to work. To address this problem, a separate linked list is used to keep track of all `stat` operations on the non-existent files (the same technique is used by RaceGuard [6]). When an `open` system call with an `O_CREAT` flag set (and no `O_EXCL` flag) is invoked on an existing file, the filename passed to the `open` is checked against the list of the non-existent filenames on the `stat` list. If there is a match between the names, an attack might be in progress and the system aborts the `open` system call with the file-already-exists error code (this is the `O_CREAT|O_EXCL` behavior).

As mentioned above, this approach deals with the filenames rather than the inodes because no inode value exists initially. Dealing with filenames in this case is not a problem since the filenames that are compared originate from the same process and are trusted to be consistent.

4.3 Data Management

To keep track of all the filesystem operations, the operation related information (process id, current time, file operation, inode and the corresponding pathname) is saved in a hash indexed by the inode value. As mentioned before, a new entry is added whenever a filesystem operation executes. The operation entries are removed by a cleaner routine that traverses the hash every second and removes all stale operation entries. A stale entry is defined to be an entry that originated more than 2 seconds ago (or 15 seconds for a directory entry) plus the current load average. For example, a load average of 1.58 causes the timeout to be 2+1.58=3.58 seconds or 15+1.58=16.58 seconds for a directory entry.

4.4 Other System Calls

Besides all the filesystem calls that are intercepted, there are several non-filesystem related system calls that need to be handled—`fork`, `exec`, and `exit`.

The `fork` system call creates a new process, and it needs to be intercepted to allow for duplication of all the entries associated with the current process. Duplicating the entries with the new process id permits file sharing between the parent and the child process. For example, a parent process might `fork` a child process that deletes some temporary files created by the parent. Without duplication of the parent entries, the child process would be found to be interfering with the parent process.

While `fork` spawns off processes that the race protection system deems to be trusted by the parent process, the `exec` system call replaces the current process image; this new process image should not, in general, be

trusted. The mediated exec system call thus removes all the entries associated with the exec'ing process. In addition, to be able to handle the setuid shell script attack described in Section 2.3, the pathnames passed to the exec system call are also inserted into the hash table.

Finally, the exit system call causes the process to terminate and this allows us to remove all the hash entries associated with the terminating process.

4.5 Portability

The code was carefully designed for maximum portability. The system dependent calls are mostly hidden behind macros that greatly simplify the porting process. The parts of the code that are OS dependent are kernel module details, system call hooking, pathname to inode resolution and locking.

To simplify the porting process even further, the code for the majority of the mediated file system calls is automatically generated by a perl script. The script automatically generates all the function declarations, all the necessary data structures as well as the function code itself. Thus the majority of the OS dependent code can be ported by modifying a small perl script.

Finally, there is nothing inherent in the design of the race protection system that prevents it from being ported to non-UNIX operating systems such as Windows.

5 Security Testing

To test the effectiveness of the race protection system, four attack scenarios were developed and carried out with the race protection being disabled and then enabled. All the attacks were successfully detected and stopped (with both default allow and default deny policies).

All of the attacks, except for the temporary file vulnerability, involve the attacker process being suspended whilst the victim process continues to run and closes the vulnerability window. The temporary file race vulnerability involves failing the open system call and forcing the victim process to deal with the consequences.

This section illustrates the attack against a race condition in using the access system call. The rest of the attacks are described in [16].

Figure 6 illustrates a successful race condition attack against a sample victim program which contains the vulnerable access code introduced in Section 2.1. The figure presents a time line with events on the left as seen by the victim and events on the right as seen by the attacker.

In step 1, the victim program is executed and it prints out the information message after calling the access system call. The victim process then sleeps for several seconds to simplify the timing attack. Step 2 illustrates the attack taking place in between the access and open system calls. Some time after the attack takes place, a victim process wakes up, opens the /tmp/race file and prints out its content which happens to be the /etc/passwd file setup by the attacker. If the victim program wrote to the file instead of reading it, the result might have been a complete system compromise. The same attack is effective against real world programs that exhibit this flaw even without the artificial presence of sleep calls.

Figure 7 illustrates the same attack taking place with the race protection system in place and the attack being thwarted.

Similarly to the previous example, the same vulnerable victim process is executed. This time though, the race protection system does not allow the /tmp/race file to be removed in step 2 since it would violate the filesystem namespace integrity (in other words, the ruleset dictates that an access operation is not allowed to be followed by an unlink). Instead, the attacker initiated process is suspended while a victim process gets a chance to finish its file operations without the attacker's interference. After five seconds (delay = base 2 seconds + the system load average of 3 = 5), the attacker process is allowed to proceed and the /tmp/race file is replaced with a symbolic link without any consecutive security side-effects.

6 Compatibility Testing

In addition to the security tests described above, the race protection system (with both default allow and default deny policies) was used on several desktop and server machines for a period of several weeks. After initial adjustments to the default deny policy, no false positives or false negatives were experienced under realistic work loads.

The race protection system was also stress tested by running multiple OpenBSD kernel compile processes in parallel with a locate.updatedb application that in-

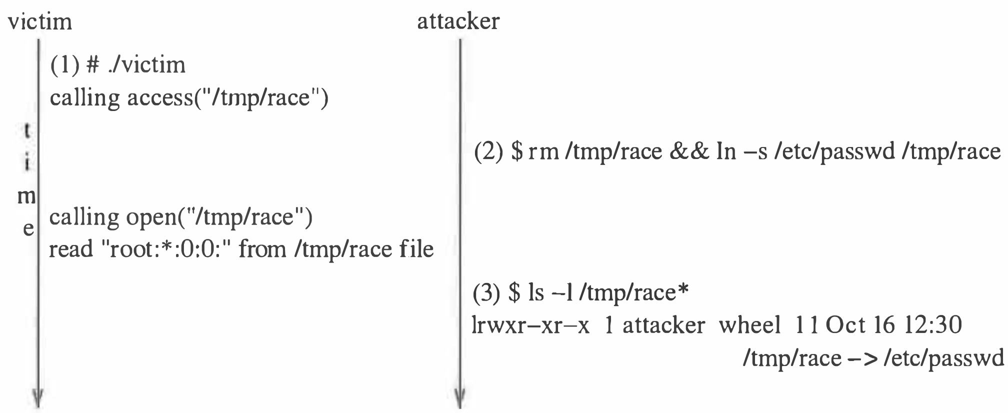


Figure 6: Successful Access Operation Attack

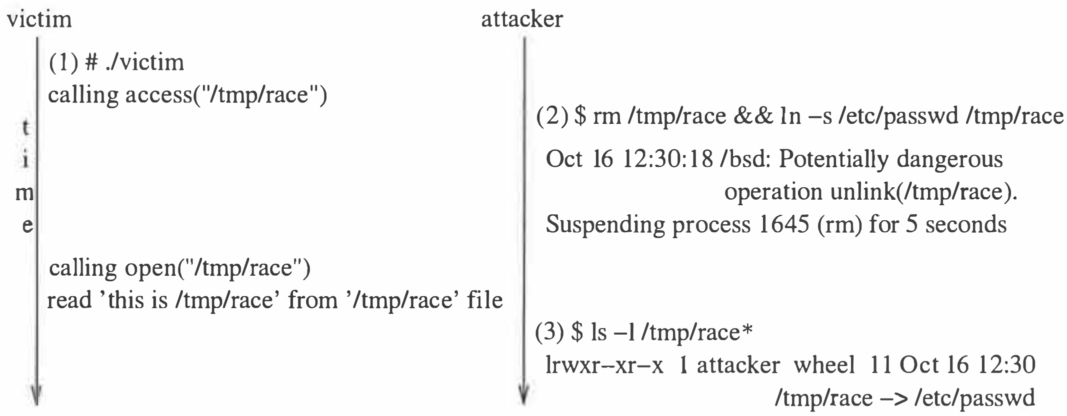


Figure 7: Unsuccessful Access Operation Attack

vokes `stat` on all the existing files. We observed no false negatives under these high loads.

This compatibility testing showed that there are no ill effects since false positives do not usually arise. Note, however, that even if false positives did arise, the only impact would be a delay in the execution of the process that was erroneously identified as an attacker.

7 Performance

To measure the overhead of the race protection system, several benchmarks were carried out on a system where the race protection was switched on and off. The tests were carried out on an OpenBSD 3.2 system running on a 1 GHz AMD Athlon computer with 256 megabytes of RAM and a 60Gig Western Digital hard drive with 8.9 ms access time.

7.1 Microbenchmarks

The first measured overhead is that of individual mediated system calls. To measure the load, each system call was called 10,000 times in a tight loop. Each test was carried out 10 times and the numbers were averaged. Table 1 shows the performance results.

As expected, the mediated system calls that need to perform path processing and check for any possible race conditions have a high overhead. For the `open` system call, the overhead of that processing amounts to over 100%. The `stat` system call does not perform any path processing and has an overhead of only 3%. The `fork` system call takes a long time to execute and needs to do little race condition processing; it exhibits relatively little overhead.

While 100% overhead of some mediated system calls might seem like an unacceptable solution, the next section illustrates that the overall application overhead is

System Call	open	stat	fork
Race Protection Off, microseconds	2.55	3.28	86.17
Race Protection On, microseconds	5.69	3.38	86.21
Total CPU Overhead (%)	123	3	0

Table 1: Microbenchmark Performance Results

	Real Time	User Time	System Time
Race Protection Off (sec)	427	363	37
Race Protection On (sec)	436	363	43
Total Overhead (%)	2	0	16

Table 2: Macrobenchmark Performance Results

much lower.

7.2 Macrobenchmarks

To measure the overall overhead, the OpenBSD kernel compile process was used as the benchmark. The compile process is computationally expensive but it also involves processing several thousand source and header files as well as creating and destroying several thousand processes. Therefore the benchmark represents a realistic workload of a loaded server.

Table 2 presents the performance numbers. As expected, the code executing in user mode shows no overhead. The code executing in kernel mode incurs a 16 percent overhead induced by the race protection system. The kernel overhead, however, is amortized over the overall benchmark execution leading to a much lower total overhead of 2 percent.

The total dynamic memory consumption initially peaks at 400Kb when the make process touches a large number of files at the same time. After the initial peak, the average memory consumption stays around 3 Kb for the rest of the benchmark process.

8 Related Work

A great deal of TOCTTOU research to date has focused on developing static tools and models for detecting race conditions at compile time. While several runtime solutions were proposed to resolve some of the problems associated with static tools, they have their own limitations.

8.1 Static Analysis

One of the first known static tools for detecting security flaws at compile time was developed as a result of the Protection Analysis project initiated at ISI by ARPA IPTO in the mid-1970's [3]. The goal of the project was to understand application and operating system security vulnerabilities and to identify "automatable pattern-directed techniques" for detecting security vulnerabilities. One of the flaws identified by the project was the "improper change" (TOCTTOU) flaw which occurred as a result of a parameter changing unexpectedly.

Another security project undertaken in the 1970's was RISOS (Research Into Secure Operating Systems) [1]. The RISOS project was a study of computer vulnerabilities in operating systems. One of the identified flaws was "inadequate serialization" which corresponds to a race condition between the access and open system calls as outlined in Figure 1.

Bishop and Dilger at UC Davis were one of the first to focus on race conditions in filesystem accesses [5]. They developed a static analysis tool for finding race conditions in C code. Their tool does not perform alias analysis nor is it control-flow aware. As a result of this, the tool can produce false negative and false positive results.

Another similar tool developed by Anguiano is based on ASTLOG [2]. Other static tools for detecting security vulnerabilities, among them race conditions, are ITS4 [17], Flawfinder [18] and RATS [15].

While the static source code analysis of TOCTTOU bugs is a valid approach, it suffers from several drawbacks. First, static tools do not have access to the runtime information needed to resolve any possible ambiguities. Filenames and environment information are not always known at compile time and even alias analysis and other proposed compiler techniques cannot solve this problem. Second, static tools are known to produce false positive as well as some false negative results [5, 2]. The reason for this being imprecise code analysis techniques as well as the absence of runtime information. Finally, the existing tools require access to the source code, which is not always available. Even if the source code is available, the time needed to run the code through one of the static tools and manually analyze the results, makes the static analysis approach less than satisfactory when trying to secure modern day systems.

8.2 Dynamic Analysis

One of the first runtime solutions was proposed by Bishop [4]. It requires applications to be modified to call a new library function that determines if a file object is safe to use. This approach is not transparent to applications and cannot be easily used to secure systems.

Another runtime solution, proposed by Solar Designer, addresses the problem by limiting users from following untrusted symbolic links created in certain directories and by limiting users from creating hard links to files they don't have read and write access to [11]. While this approach might be effective in stopping some race condition attacks, it is too limited to stop other variations of attacks. In addition, this approach breaks the applications that depend on the behavior restricted by the solution.

Similarly, RaceGuard focuses on race conditions that occur during the creation of temporary files [6]. While this is a valid approach, it is limited to temporary file race vulnerabilities.

9 Future Work

While we tried to address all known practical race conditions, there are still some questions left unanswered.

First, our implementation does not strictly enforce the correct behavior of allowing a victim process to proceed first and close its race condition window. The heuristics of dynamically calculating the delay performed well in our lab tests but this is no guarantee that they will work in all scenarios. Our delay formula and hash table cleaning times were experimentally created using ad-hoc methods. A proper justification of these, via an analysis of programmer isolation expectations and process runtimes, remains to be provided. Future implementations should address this issue more carefully.

Second, our implementation does not address the issue of subdirectory locking. As described in Section 2.2, attackers can move directories around thereby tricking processes into accessing arbitrary files. Future implementations should analyze this threat better for all race condition cases and not just the case described in Section 2.2.

Third, our implementation does not address the following race condition attack. Suppose an old style shell program that does not implement `test` functionality as a built-in function is used to interpret a script. (Alternatively, a script could be written to explicitly use `/bin/test` rather than the built-in function.) The shell `fork`'s a subprocess which `exec`'s the `test` program. The subprocess then invokes the system call `stat` and communicates its result to the shell via the process exit status. Since the subprocess has exited, the hash table entries pertaining to the file will be deleted by the `exit` system call mediation code. No hash entry would correspond to the parent process, since the shell only handles the file as a string. Based on the result of the `stat`, e.g., in a command such as `/bin/test -r $file || foo > $file`, another subprocess uses the file. This is a scenario that should be rare, but is nevertheless not detected by our scheme. Note that the RaceGuard scheme [6] is unable to protect against this as well. Better analysis and improved heuristics are required to detect this kind of pseudo-transaction.

Finally, our implementation only deals with timing races—i.e., race conditions that exist for short periods of time in between two file system call invocations. We do not handle other types of race conditions such as storage races. An example of a storage race condition is a predictable `/tmp` filename attack which involves an attacker planting a malicious link in the `/tmp` directory. At a later time, a victim process follows the link and clobbers an arbitrary file. Future implementations should try to classify all existing race condition types and ways to deal with them.

10 Conclusions

Race conditions in filesystem accesses occur when applications incorrectly assume that a sequence of filesystem operations is isolated and the filesystem namespace is otherwise static. Even though the existence of the TOCTTOU bugs dates back to almost 30 years, new race conditions are still being discovered today.

To prevent race conditions, a system should provide an illusion of an immutable namespace existing without external interferences. This goal resembles that of a transaction enabled system where the properties of atomicity, consistency, isolation and durability have to be satisfied. Since mainstream operating systems are not designed to satisfy any of the ACID properties, the race protection system has to emulate certain features to achieve the desired effect. A system that creates an illusion of an immutable namespace should satisfy the isolation property and should be effective in stopping race conditions. An immutable namespace effect is achieved by detecting and stopping race conditions. Race conditions are detected by tracking all file-related system activity and identifying any filesystem operation sequences that could possibly result in a TOCTTOU condition. When two filesystem operations are found to be interfering with each other, the one to access a file object last is suspended. This allows the first operation to proceed without any interferences. This solution stops all file race conditions attacks known to us and is also designed to prevent future attacks. It does this with minimum performance overhead while retaining application compatibility.

11 Acknowledgments

We would like to thank Christina Martinez for proofreading the paper, Peter Bartoli and Marshall Beddoe for providing us with hardware testbeds, Crispin Cowan for shepherding this paper and Keith Marzullo, Stefan Savage and the anonymous reviewers for their comments and suggestions.

This work is supported in part by the Office of Naval Research, Award N00014-01-1-0981. The opinions expressed in this paper are those of the authors.

References

[1] R. P. Abbott, J. S. Chin, J. E. Donnelley, W. L. Konigsford, S. Tokubo, and D. A. Webb. Security Analysis and Enhancements of Computer Operating Systems. NBSIR 76–1041, Institute for Computer Sciences and Technology, National Bureau of Standards, April 1976.

[2] R. Anguiano. A Static Analysis Technique for the Detection of TOCTTOU Vulnerabilities. Master Thesis, University of California Davis, 2001.

[3] R. Bisbey and D. Hollingsworth. Protection Analysis Project Final Report. ISI/RR-78-13, DTIC AD A056816, USC/Information Sciences Institute, May 1978.

[4] M. Bishop. Race Conditions, Files, and Security Flaws: or, The Tortoise and the Hare Redux. Technical Report 95-8, Department of Computer Science, University of California at Davis, September 1995.

[5] M. Bishop and M. Dilger. Checking for Race Conditions in File Accesses. Computing systems, pages 131–152, Spring 1996.

[6] C. Cowan, S. Beattie, C. Wright, and G. Kroah-Hartman. RaceGuard: Kernel Protection From Temporary File Race Vulnerabilities. In Proceedings of the 10th USENIX Security Symposium, August 2001.

[7] C. Cowan. Personal communication.

[8] T. Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In Proceedings of the Network and Distributed Systems Security Symposium, February 2003.

[9] G. Guninski. OpenBSD 2.9,2.8 local root compromise. Bugtraq mailing list, June 2001.

[10] The OpenBSD Project. `http://www.openbsd.org/`

[11] Openwall Project. Linux kernel patch. `http://www.openwall.com/linux/`

[12] N. Provos. Improving Host Security with System Call Policies. CITI Technical Report 02-3, November 2002.

[13] W. Purczynski. rm - recursive directory removal race condition. Bug-fileutils mailing list, March 2002.

[14] F. Schmuck and J. Wyllie. Experience with transactions in QuickSilver. In Proceedings of the 13th ACM Symposium on Operating System Principles, pages 239–53, October 1991.

[15] Secure Software. Rough Auditing Tool for Security (RATS). `http://www.securesoftware.com/rats.php`

[16] E. Tsyrklevich. Dynamic Detection and Prevention of Race Conditions in File Accesses. Master Thesis, University of California San Diego, 2002.

[17] J. Viega, J. Bloch, T. Kohno, and G. McGraw. ITS4: a static vulnerability scanner for C and C++ code. In proceedings of 16th Annual Computer Security Applications Conference, December 2000.

[18] D. Wheeler. Flawfinder. `http://www.dwheeler.com/flawfinder`

Improving Host Security with System Call Policies

Niels Provos
Center for Information Technology Integration
University of Michigan
provos@citi.umich.edu

Abstract

We introduce a system that eliminates the need to run programs in privileged process contexts. Using our system, programs run unprivileged but may execute certain operations with elevated privileges as determined by a configurable policy eliminating the need for suid or sgid binaries. We present the design and analysis of the "Systrace" facility which supports fine grained process confinement, intrusion detection, auditing and privilege elevation. It also facilitates the often difficult process of policy generation. With Systrace, it is possible to generate policies automatically in a training session or generate them interactively during program execution. The policies describe the desired behavior of services or user applications on a system call level and are enforced to prevent operations that are not explicitly permitted. We show that Systrace is efficient and does not impose significant performance penalties.

1 Introduction

Computer security is increasing in importance as more business is conducted over the Internet. Despite decades of research and experience, we are still unable to make secure computer systems or even measure their security.

We take for granted that applications will always contain exploitable bugs that may lead to unauthorized access [4]. There are several venues that an adversary may choose to abuse vulnerabilities, both locally or remotely. To improve the security of a computer system, we try to layer different security mechanisms on top of each other in the hope that one of them will be able to fend off a malicious attack. These layers may include firewalls to restrict network access, operating system primitives like non-executable stacks or application level protections like privilege separation [30]. In theory and practice, security increases with the number of layers that need to be circumvented for an attack to be successful.

Firewalls can prevent remote login and restrict access, for example to a web server only [12]. However, an adversary who successfully exploits a bug in the web server and gains its privileges may possibly use them in subsequent attacks to gain even more privileges. With local access to a system, an adversary may obtain *root* privileges, *e.g.*, by exploiting *setuid* programs [5, 11], using *localhost* network access or special system calls [8].

To recover quickly from a security breach, it is important to detect intrusions and to keep audit trails for post-mortem analysis. Although there are many intrusion detection systems that analyze network traffic [27] or host system activity [21] to infer attacks, it is often possible for a careful intruder to evade them [31, 36].

Instead of detecting intrusions, we may try to confine the adversary and limit the damage she can cause. For filesystems, access control lists [15, 32] allow us to limit who may read or write files. Even though ACLs are more versatile than the traditional Unix access model, they do not allow complete confinement of an adversary and are difficult to configure.

We observe that the only way to make persistent changes to the system is through *system calls*. They are the gateway to privileged kernel operations. By monitoring and restricting system calls, an application may be prevented from causing harm. Solutions based on system call interposition have been developed in the past [20, 24]. System call interposition allows these systems to detect intrusions as policy violations and prevent them while they are happening. However, the problem of specifying an accurate policy still remains.

This paper presents *Systrace*, a solution that efficiently confines multiple applications, supports multiple policies, interactive policy generation, intrusion detection and prevention, and that can be used to generate audit logs. Furthermore, we present a novel approach called *privilege elevation* that eliminates the need for *setuid* or *setgid* binaries. We discuss the design and implementation of Systrace and show that it is an extensible and efficient solution to the host security

problem.

The remainder of the paper is organized as follows. Section 2 discusses related work. In Section 3, we provide further motivation for our work. Section 4 presents the design of Systrace and Section 5 discusses its implementation. We present an analysis of the system in Section 6. In Section 7, we present a detailed performance analysis of our system. We discuss future work in Section 8 and conclude in Section 9.

2 Related Work

Although capabilities [26] and access control lists [15, 32] extend the traditional Unix access model to provide finer-grained controls, they do not prevent untrusted applications from causing damage. Instead, we may use mechanisms based on system call interception or system call interposition to prevent damage from successful intrusions.

Janus, by Goldberg *et al.* [20], is one of the first system call interception tools. It uses the *ptrace* and */proc* mechanisms. Wagner states that *ptrace* is not a suitable interface for system call interception, *e.g.*, race conditions in the interface allow an adversary to completely escape the sandbox [37]. The original Janus implementation has several drawbacks: Applications are not allowed to change their working directory or call *chroot* because Janus cannot keep track of the application's changed state. Janus has evolved significantly over time and its latest version uses a hybrid approach similar to Systrace to get direct control of system call processing in the operating system [18].

One particularly difficult problem in application confinement is symlinks, which redirect filesystem access almost arbitrarily. Garfinkel introduces safe calling sequences that do not follow any symlinks [18]. The approach uses an extension to the *open* system call that is specific to the Linux operating system but breaks any application that accesses filenames containing symlinks. Systrace solves this problem using filename normalization and argument replacement. Currently, Janus does not address intrusion detection, auditing or policy generation.

Jain and Sekar [24] offer another fairly complete treatment of system call interposition. On some systems their implementation is based on *ptrace* and suffers the problems mentioned above. Furthermore, they do not address the problem of naming ambiguities that may result in policy circumvention. Because C++ is used as their policy language, creating comprehensive policies is difficult. Systrace, on the other hand, supports automatic and interactive policy generation which allows us to create policies quickly even in very complex environments.

Other systems that use mechanisms like system call interception are BlueBox [10], Cerb [14], Consh [2], MAPbox [1] and Subterfugue [13].

Peterson *et al.* present a general-purpose system call API for confinement of untrusted programs [28]. The API is flexible but has no provisions for recording audit trails or intrusion detection. Furthermore, specifying security policies is labor intensive as the sandbox needs to be programmed into applications.

Domain Type Enforcement [3, 38] is a kernel-level approach to restrict system access for all processes depending on their individual domains. A complete DTE implementation requires extensive changes to the operating system and does not automatically extend to new subsystems. Because policies are locked down on system start, users may not create individual policies. In contrast to Systrace, DTE domains do not differentiate between users. We feel that system call interposition offers higher flexibility as it allows us to design and create a simple system that also addresses policy generation, audit trails, intrusion detection, etc.

The security architecture for the Flask microkernel emphasizes *policy flexibility* and rejects the system call interception mechanism claiming inherent limitations that restrict policy flexibility [34]. Instead, the Flask system assigns security identifiers to every object and employs a security server for policy decisions and an object server for policy enforcement. However, Systrace uses a hybrid design that allows us to overcome the traditional limitations of system call interception; see Section 6.

SubOS [23] takes a similar approach based on object labeling to restrict access to the system. Depending on their origin, objects are assigned sub-user identifiers. A process that accesses an object inherits its sub-user id and corresponding restrictions. As a result, a process subverted by a malicious object may cause only limited damage. In practice, there are only a few applications that can be subverted that way and enforcing security policies for these applications is sufficient to prevent malicious data from causing damage.

Forrest *et al.* analyze system call sequences to discriminate between processes [16]. Their work is extended by Hofmeyer *et al.* to achieve intrusion detection by recording the system calls that an application executes and comparing the recorded sequences against a database of good sequences [21]. Abnormal sequences indicate an ongoing intrusion. The training process that collects good system call sequences is similar to the automatic policy generation feature of Systrace. Wespi *et al.* further extend this approach by using

variable-length patterns to match audit events [39]. Although analyzing system call or audit sequences is an effective mechanism to detect intrusions, it does not help to prevent them. Recent research also shows that *mimicry* attacks can evade intrusion detection system based on system call sequences [35, 36]. Systrace not only detects such intrusions, it can also prevent them or at least limit the damage they can cause. Furthermore, evasion attacks are not possible as we discuss in Section 6.

3 Motivation and Threat Model

Most applications that run on computer systems are too complex and complicated to trust: web browsers, name servers, etc. Even with access to the source code, it is difficult to reason about the security of these applications. They might harbor malicious code or contain bugs that are exploitable by carefully crafted input.

Because it is not possible to find all vulnerabilities, we assume the existence of programming errors known to the adversary that she can use to gain unauthorized access to the system.

We limit the impact an adversary can have on the system by restricting the operations an application is allowed to execute. The observation that changes relevant to security are performed via *system calls* makes the enforcement of restrictions at the system call level a natural choice.

An application is confined by a set of restrictions which are expressed by a security policy. Defining a correct policy is difficult and not possible without knowing all possible code paths that an uncompromised application may take. Therefore we require the policy language to be intuitive while still expressive. It should be possible to generate policies without complete knowledge of an application.

We may use the security policy as a specification that describes the expected behavior of an application. When monitoring the operations an application attempts to execute, any deviation from the specified policy indicates a security compromise [25]. To further facilitate forensic analysis of an intrusion, we also wish to generate an audit log of previous operations related to the application.

Experience shows that adversaries escalate their privileges by abusing *setuid* or *setgid* programs [5]. These programs are executed by the operating system with different privileges than the user starting them. Although increasing privileges is often necessary for correct operation, the *setuid* model is too coarse grained. We aim to provide a fine-grained model that eliminates the need for *setuid* binaries and integrates a method to elevate privilege into a policy language.

Systrace realizes these goals and is an effective improvement of host security that limits the damage an adversary can cause by exploiting application vulnerabilities. The next section discusses the design of Systrace.

4 Design

There are several approaches for implementing system call interposition. We may use existing interception mechanisms to create an implementation completely in user space, implement the system entirely at the kernel-level, or choose a hybrid of both. A user space implementation is often more portable but may suffer a larger performance impact. Furthermore, the interception mechanism may not provide the required security guarantees or may make it difficult to keep track of operating system state like processes exiting and forking. A notable exception is SLIC [19], a mechanism to create extensible operating systems via system call interposition. Unfortunately, it is not portable and adds significant complexity to the operating system.

On the other hand, an implementation completely at the kernel-level is likely to be fast but less portable and also causes a significant increase in the complexity of the operating system.

We choose a hybrid approach to implement a small part of the system at the kernel-level. The kernel-level part supports a fast path for system calls that should always be allowed or denied. That case should incur almost no performance penality because it does not require a context switch to ask a user space policy daemon for a decision.

Some control in the kernel also allows us to make the system *fail-safe*, *i.e.*, no application can escape its sandbox even if there are unforeseen errors that might cause the monitor itself to terminate. When the sandboxing process terminates, the kernel terminates all processes that it was monitoring. Additionally, the kernel keeps track of creation of new processes and of processes that exit. Child processes inherit the policy of their parent.

If the kernel cannot use the fast path for a system call, it must ask the policy daemon in user space for a policy decision. In that case, the process is blocked until the daemon returns with an answer to permit the system call or to deny it with a certain error code. Information is exported from the kernel to user space via a simple yet comprehensive interface.

The user space policy daemon uses the kernel inter-

face to start monitoring processes and to get information about pending policy decisions or state changes. The state changes may be process creation, processes exiting, processes changing uid or gid, and other state changes.

The daemon may also request information about the result of a system call. This allows us to know, for example if the *execve* system call has succeeded in replacing the current process image with a new application. This event can install a new policy from the policy database.

System call interception does not provide atomicity between the time a policy decision is made and the time a system call is executed, *i.e.* the time of check is not the time of use (TOCTOU). As a result,
sary can change the system call before it is executed but after the policy daemon has inspected it. For example, two processes that share parts of their address space may cooperate to present one set of system call arguments to the policy daemon and another one to the kernel. When the kernel suspends the first process to consult the policy daemon, the second process is still running and may change the system call arguments of the first process after they have been inspected by the daemon. For filesystem access, an adversary may also redirect the access by changing a component in the filename to a symbolic link after the policy check. This lack of atomicity may allow an adversary to escape the sandbox.

We prevent these race conditions by replacing the system call arguments with the arguments that were resolved and evaluated by Systrace. The replaced arguments reside in kernel address space and are available to the monitored process via a read-only look-aside buffer. This ensures that the kernel executes only system calls that passed the policy check.

Before making a policy decision, the system call and its arguments are translated into a system independent human-readable format. The policy language operates on that translation and does not need to be aware of system call specific semantics.

4.1 Policy

Existing frameworks for making policy decisions propose generic policy languages [6, 7] and provide policy evaluation methods but are more complex than necessary in our case. For that reason, we create our own policy language and evaluator. This approach has also been taken by other sandboxing tools [1, 20].

We use an ordered list of policy statements per system call. A policy statement is a boolean expression B combined with an action clause: B then *action*. Valid actions are *ask*, *deny* or *permit* plus optional flags. If the boolean expression evaluates to *true*, the specified action is taken. The *ask* action requires the user to deny or permit the system call explicitly.

A boolean expression consists of variables X_n and the usual logical operators: *and*, *or* and *not*. The variables X_n are tuples of the form (*subject op data*), where *subject* is the translated name of a system call argument, *data* is a string argument, and *op* a function with boolean return value that takes *subject* and *data* as arguments.

The set of all lists forms the security policy. For a given system call, policy evaluation starts at the beginning of the system call specific list and terminates with the first boolean expression that is true; see Figure 1. The action from that expression determines if the system call is denied or allowed.

If no boolean expression becomes true, the policy decision is forwarded to the user of the application or automatically denied depending on the configuration. Section 4.2 explains in more detail how this mechanism is used to generate policies interactively or automatically. When denying a system call, it is possible to specify which error code is passed to the monitored application.

To create comprehensive policies that apply to different users, policy statements may carry predicates. A policy statement is evaluated only if its predicate matches and ignored otherwise. Using predicates, it is possible to restrict the actions of certain users or be more permissive with others, for example system administrators. Predicates are appended to the policy statement and are of the form *if* {*user,group*} *op data*, where *op* is either equality or inequality and *data* a user or group name.

The *log* modifier may be added to a policy statement to record matching system calls. Every time a system call matches this policy statement, the operating system records all information about the system call and the resulting policy decision. This allows us to create arbitrarily fine-grained audit trails.

4.2 Policy Generation

Creating policies is usually relegated to the user who wishes to sandbox applications. Policy generation is not an easy task as some policy languages resemble complicated programming languages [24]. Although those languages are very expressive, the difficulty of creating good policies increases with the complexity of the policy language.

Our definition of a *good policy* is a policy that allows only those actions necessary for the intended function-

```
Policy: /usr/sbin/named, Emulation: native
     native-__sysctl: permit
     native-accept: permit
     native-bind: sockaddr match "inet-*:53" then permit
     native-break: permit
     native-chdir: filename eq "/" then permit
     native-chdir: filename eq "/namedb" then permit
     native-chroot: filename eq "/var/named" then permit
     native-close: permit
     native-connect: sockaddr eq "/dev/log" then permit
                    ...
```

Figure 1: Partial policy for the name daemon. Policies can be improved iteratively by appending new policy statements. The policy statement for *bind* allows the daemon to listen for DNS requests on any interface.

ality of the application but that denies everything else.

Clearly, we can construct a policy that matches our definition by enumerating all possible actions that an application needs for correct execution. If an action is not part of that enumeration, it is not allowed.

In the following, we show how our policy language facilitates policy construction. The policy language is designed to be simple. Each policy statement can be evaluated by itself, thus it is possible to extend a policy by appending new policy statements. The major benefit of this approach is that a policy can be generated iteratively.

We create policies automatically by running an application and recording the system calls that it executes. We translate the system call arguments and canonically transform them into policy statements for the corresponding system calls. When an application attempts to execute a system call during the training run, it is checked against the existing policy and if not covered by it, a new policy statement that permits this system call is appended to the policy. Unlike intrusion detection systems that analyze only sequences of system call names [16, 21], our policy statements capture the complete semantics of a system call and are not subject to evasion attacks [36].

On subsequent runs of the application, the automatically created policy is used. For some applications that create random file names, it is necessary to edit the policies by hand to account for nondeterminism.

When generating policies automatically, we assume that the application itself does not contain malicious code and that it operates only with benign data. Otherwise, the resulting policies might permit undesirable actions.

To address cases for which our assumptions do not hold or for which it is impossible to exercise all code paths in a training run, we use interactive policy generation. Interactivity implies a user needs to make policy decisions when the current policy does not cover the attempted system call. When a policy decision is required by the user, she is presented with a graphical notification that contains all relevant information; see Figure 2. She then either improves the current policy by appending a policy statement that covers the current system call, terminates the application, or decides to allow or deny the current system call invocation.

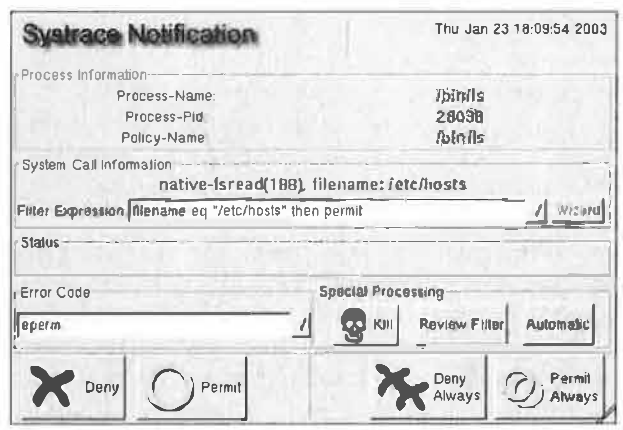


Figure 2: A graphical notification assists the user when a policy decision is required. A user may decide to allow or deny the current system call or to refine the policy.

If we do not exercise all possible code paths, automatic policy generation does not enumerate all legitimate actions of an application and by itself is not sufficient to create a *good* policy. However, it provides a base policy that covers a subset of necessary actions. In conjunction with interactive policy generation, we iteratively refine the policy by enumerating more valid actions until the policy is good.

The system assists the user by offering generic policy templates that can be used as a starting point. Once an initial policy has been created, policy notifications appear only when an attempted operation is not covered by the configured policy. This might indicate that

a new code path is being exercised, or that a security compromise is happening. The user may either permit the operation or deny and investigate it.

Once a security policy for an application has been finalized, automatic policy enforcement may be employed. In that case, the user is not asked for a policy decision when an application attempts to execute a system call that is not covered by the policy. Instead, the system call is denied and an error code returned to the application. The errant attempt is logged by the operating system.

4.3 Privilege Elevation

Beyond restricting an application to its expected behavior, there are situations in which we would like to increase its privilege. In Unix, there are many system services and applications that require *root* privilege to operate. Often, higher privilege is required only for a few operations. Instead of running the entire application with special privilege, we elevate the privilege of a single system call. The motivation behind *privilege elevation* is the principle of least privilege: every program and every user should operate using the least amount of privilege necessary to complete the job [33].

To specify that certain actions require elevated privilege, we extend the policy language to assign the desired privilege to matching policy statements. Systrace starts the program in the process context of a less privileged user and the kernel raises the privilege just before the specified system call is executed and lowers it directly afterwards.

As every user may run their own policy daemon, privilege elevation is available only when the Systrace policy daemon runs as *root*. Otherwise, it would be possible for an adversary to obtain unauthorized privileges by creating her own policies. Identifying the privileged operations of *setuid* or *setgid* applications allows us to create policies that elevate privileges of those operations without the need to run the whole application at an elevated privilege level. As a result, an adversary who manages to seize control of a vulnerable application receives only very limited additional capabilities instead of full privileges.

The *ping* program, for example is a *setuid* application as it requires special privileges to operate correctly. To send and receive ICMP packets, *ping* creates a raw socket which is a privileged operation in Unix. With privilege elevation, we execute *ping* without special privileges and use a policy that contains a statement granting *ping* the privilege to create a raw socket.

Unix allows an application to discard privileges by changing the *uid* and *gid* of a process. The change is permanent and the process cannot recover those privileges later. If an application occasionally needs special privileges throughout its lifetime dropping privileges is not an option. In this case, privilege elevation becomes especially useful. For example, the *ntpd* daemon synchronizes the system clock. Changing system time is a privileged operation and *ntpd* retains *root* privileges for its whole lifetime. A recent remote root vulnerability [17] could have been prevented with single system call privilege elevation.

5 Implementation

We now give an overview of the Systrace implementation. Systrace is currently available for Linux, Mac OS X, NetBSD, and OpenBSD; we concentrate on the OpenBSD implementation.

To help reason about the security of our implementation, simplicity is one of our primary goals. We keep the implementation simple by introducing abstractions that separate different functionalities into their own components. A conceptual overview of the system call interception architecture is shown in Figure 3.

When a monitored application executes a system call, the kernel consults a small in-kernel policy database to check if the system call should be denied or permitted without asking the user space daemon. At this point, policy decisions are made without inspecting any of the system call arguments. Usually, system calls like *read* or *write* are always permitted. The kernel communicates via the `/dev/systrace` device to request policy decisions from the daemon.

While processes may have different policies, the initial policy for all system calls defers policy decisions to a corresponding user space daemon. When the kernel is waiting for an answer, it suspends the process that requires a policy decision. If the process is awakened by a signal before a policy decision has been received, the kernel denies the current system call and returns an error. To enforce synchronization, each message from the kernel carries a sequence number so that answers from user space can be matched against the correct message. The sequence number ensures that a user space policy decision is not applied to a system call other than the one that caused the message.

When the user space policy daemon receives a request for a decision, it looks up the policy associated with the process and translates the system call arguments. To translate them, we register translators for

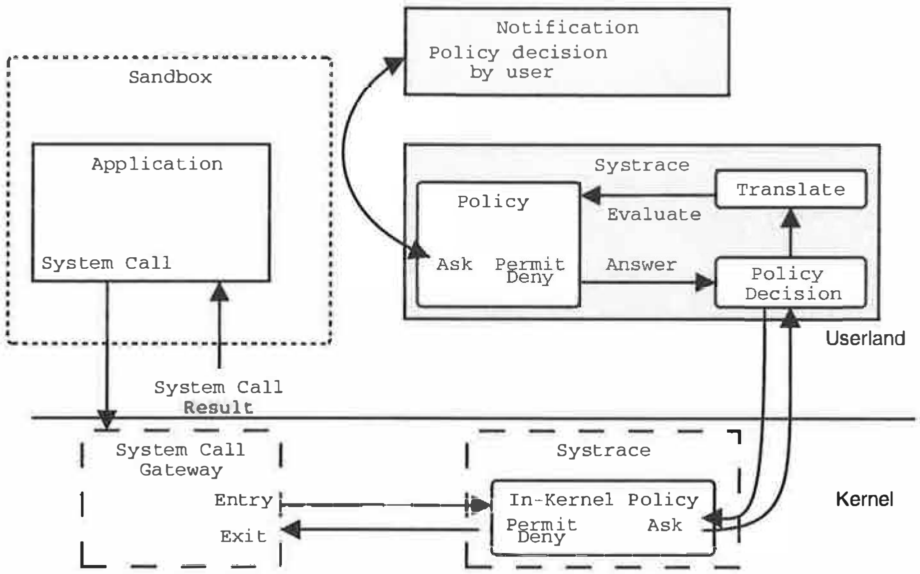


Figure 3: Overview of system call interception and policy decision. For an application executing in the sandbox, the system call gateway requests a policy decision from Systrace for every system call. The in-kernel policy provides a fast path to permit or deny system calls without checking their arguments. For more complex policy decisions, the kernel consults a user space policy daemon. If the policy daemon cannot find a matching policy statement, it has the option to request a refined policy from the user.

each argument in a system call. The translation of the

```
socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
```

system call takes the following form:

```
socket: sockdom: AF_INET, socktype: SOCK_RAW
```

The third argument has not been translated because it is irrelevant on the supported Unix systems.

While many argument translators are fairly simple, translating filenames is more complicated. Filenames in Unix are relative to the current working directory of a process. In order to translate a filename into an unambiguous absolute path name, we need to know the current working directory of the monitored application even if it is working in a *chroot* environment. Additionally, all symbolic links in components of the filename need to be resolved so that access restrictions imposed by policy cannot be circumvented by an adversary[1].

The translators also act as *argument normalizers*. The argument replacement framework is used to replace the original arguments with their translation. As the kernel sees only normalized arguments, an adversary cannot use misleading arguments to circumvent a security policy. The kernel makes the rewritten arguments available to the monitored process via a look-aside buffer before resuming execution of the system call. Furthermore, we disallow the process to follow any symbolic links because no component of a normalized filename contains symbolic links that should be followed.

A policy statement that permits the creation of raw sockets might look like this:

```
socket:  socktype eq "SOCK_RAW" then permit
```

The operators in the boolean expression use the translated human-readable strings as input arguments. We currently support `eq`, `match`, `re` and `sub` as operators:

- The `eq` operator evaluates to *true* only if the system call argument matches the text string in the policy statement exactly.
- The `match` operator performs file name globbing as found in the Unix shell. It can be used to match files in directories for file name arguments.
- The `re` operator uses regular expressions to match system call arguments. It is very versatile but more expensive to evaluate than other operators.
- The `sub` operator evaluates to *true* only if the system call argument contains the specified substring.

If evaluating the policy for the current system call results in either *deny* or *permit*, the policy daemon re-

[1] For system calls like *lstat* or *readlink*, we resolve all but the last component which may be a symbolic link as the operating system does not follow it.

turns the answer to the kernel which then awakens the sleeping process. Otherwise, the user monitoring the applications is asked for a policy decision. The notification mechanism can be implemented independently from the rest of the system and is currently either a graphical user interface or a text prompt on the terminal. At this point, the user can add new policy statements to the policy.

Policies for system calls accessing the filesystems tend to be similar. For example, the *access*, *stat*, and *lstat* system calls all fulfill similar functionality. In order to avoid duplication of policy, we introduce *system call aliasing* to map system calls with similar functionality into a single virtual system call which is then used for policy evaluation. Currently, *fsread* is used for system calls that grant read access to filesystem objects, and *fswrite* for system calls that cause change in the filesystem. The *open* system call is mapped to *fsread* or *fswrite* depending on the kind of filesystem access that is indicated by its arguments. System call aliasing reduces the size of policies and simplifies policy generation.

It is possible to make policies more flexible by using predicates. Policy statements are only evaluated if their predicate matches. For example, to prevent *root* access via the SSH daemon, a policy statement that permits the execution of a shell could be predicated so that it applies only to non-root users. In order to keep track of a process' *uid* and *gid*, the kernel sends informational messages to the policy daemon when those values change.

The *execve* system call is treated specially. When a process executes another application, its in-memory image is replaced with the one of the executed program. To support more fine-grained policies, we can set a new policy for the process. The policy is obtained from the name of the executed application. As a result, one Systrace daemon may concurrently enforce multiple policies for multiple processes.

Policies for different applications are stored in a policy directory as separate files. Users may store their own policies in a user specific policy directory. The system administrator may also provide global policies for all users. To sandbox applications, users start them with the Systrace command line tool. Administrators may assign a Systrace login shell to users to enforce policy for all their applications.

6 Analysis

An adversary who takes control of a sandboxed application may try to escape the sandbox by confusing the policy enforcement tool and tricking it into allowing actions that violate policy. Although many sandboxing tools share common problems, we present novel solutions to some of them and discuss inherent limitations of policy systems based on system call interposition.

6.1 Security Analysis

To enforce security policies effectively by system call interposition, we need to resolve the following challenges: incorrectly replicating OS semantics, resource aliasing, lack of atomicity, and side effects of denying system calls [18, 34, 37]. We briefly explain their nature and discuss how we address them.

The sandboxing tool must track operating system state in order to reach policy decisions. Systrace, for example, must keep track of process *uids* and the filename of the program binary the monitored process is executing. To avoid incorrectly replicating OS semantics, our kernel-level implementation informs the Systrace daemon about all relevant state changes.

Resource aliasing provides multiple means to address and access the same operating system resource. For example, on some Unix systems, it is possible to gain access to files by communicating with a system service or by using symbolic links in the filesystem to create different names for the same file. An adversary may use these indirections to circumvent policy and obtain unauthorized access. The system call interposition mechanism is unaware of system services that allow proxy access to operating system resources. When creating policies that allow a sandboxed application to contact such system services, we need to be aware of the consequences. However, we can prevent aliasing via symbolic links or relative pathnames as discussed below.

Another problem is the lack of atomicity between the time of check and the time of use that may cause the mapping of name to resource to change between policy decision and system call execution. An adversary may cause such a state change that allows a process to access a different resource than the one originally approved, for example a cooperating process sharing memory may rewrite system call arguments between policy check and execution.

Systrace solves both aliasing and atomicity problems by *normalizing* the system call arguments. We provide the *normalized* values to the operating system in such a way that the name to resource mapping cannot be changed by an adversary. For filenames, this includes resolving all symbolic links and all relative paths. The only exception are system calls like *readlink*, for which we do not resolve the last component. As resolved file-

names do not contain any symbolic links that should be followed, the kernel denies the monitored process to follow any symbolic links. Instead of placing the rewritten arguments on the stack as done in MAPbox [1], we provide a read-only look-aside buffer in the kernel. Otherwise, multi-threaded applications can change system call arguments after the policy check.

As a result, evasion attacks [35, 36] are no longer possible. System calls are allowed only if their arguments match a statement in the policy and are denied otherwise.

However, we need to take side effects of denying system calls into consideration. If we assume correct security policy, system calls are denied only if an application attempts to do something that it should not. As the behavior of many applications depends on the error code returned to them, we can specify the error code as part of the Systrace policy. Every system call has its own set of valid return codes which does not always include `EINTR` or `EPERM`. To avoid confusing applications, we allow policies to set their own error codes instead of mandating a fixed value[2]. For example, we let the kernel return `EACCESS` for the *stat* system call if the application should think that it is not permitted to access a certain file. On the other hand, returning `ENOENT` causes the application to think that the file does not exist.

Furthermore, we address secure process detaching and policy switching, problems that are often overlooked. When an application executes a new program, the operating system replaces the code that the process is running with the executed program. If the new program is trusted, we may wish to stop monitoring the process that runs it. On the other hand, a new program also implies new functionality that could be confined better with a different, more suitable policy. If requested, Systrace reports the return value of a system call to indicate if it was successfully executed or not. In the case of *execve*, success indicates that the monitored process is running a new program and we allow the policy to specify if we should detach from the process or allow a different policy to take effect. After these changes take effect, the execution of the process is resumed.

Because the security of our system relies on the integrity of the filesystem, we assume that it is secure. If an adversary can control the filesystem, she may modify the policies that determine the permissible operations for monitored applications or replace trusted programs with malicious code.

[2] This does not prevent faulty applications that are written without proper error handling from misbehaving. In that case, Systrace may help to identify incorrect exception handling.

Audit trails may be generated by adding the *log* modifier to policy statements. An an example, for an audit trail of all commands a user executes, it is sufficient to Systrace her shell and log all the executions of *execve*.

The benefit of privilege elevation is the reduction of privilege an application requires for its execution. Applications that formerly required *root* privilege for their entire lifetime now execute only specific system calls with elevated privilege. Other system calls are executed with the privilege of the user who invoked the application. The semantics of *setuid* prevent a user from debugging privileged applications via *ptrace*. We apply the same semantics when policy elevates an application's privilege.

6.2 Policy generation

Policy generation is an often neglected problem. In order for a sandbox to function correctly, it requires a policy that restricts an application to a minimal set of operations without breaking its functionality. To facilitate policy generation, our policy language allows policies to be improved iteratively by appending new policy statements.

We can generate policies automatically by executing applications and recording their normal behavior. Each time we encounter a system call that is not part of the existing policy, we append a new policy statement that matches the current translated system call.

The resulting policy covers the executed code path of the application. For applications that randomize arguments, we post process the policy to make it independent of arguments with random components.

For example, when `mkstemp("/tmp/confXXXXXX")` creates the file `/tmp/confJ31A69`, automatic policy generation appends a corresponding policy statement:

```
fswrite: filename eq "/tmp/confJ31A69" then permit
```

Post processing changes the policy statement so that it is independent of the randomness and thus applies to subsequent executions of the application:

```
fswrite: filename match "/tmp/conf*" then permit
```

Automatic policy generation and the process of profiling normal application behavior by Hofmeyr *et al.* [21] face similar problems. We need to make sure that no abnormal behavior occurs during policy training and try to exhaust all possible code paths. However, interactive and automatic policy generation go hand in hand. We do not require a complete policy to sandbox an application because we may request a policy decision from the user if an operation is not covered by the

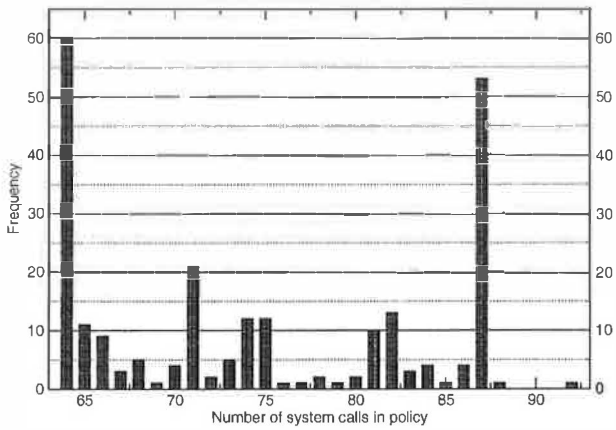


Figure 4: Analysis of the number of system calls that applications are allowed to execute. Most applications use only sixty to ninety different system calls. As average Unix systems support several hundred system calls, we disallow the execution of all other system calls to prevent an adversary from using them to cause damage. Note that the abscissa origin is not zero.

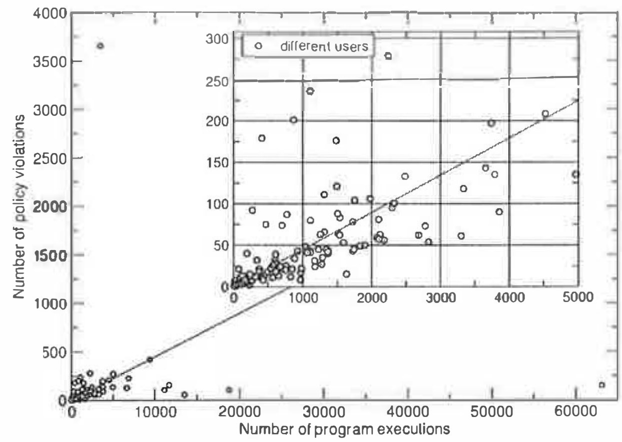


Figure 5: The cross correlation of the number of policy violations and the number of program executions allows us to identify users that exhibit unusual behavior. The user with the most policy violations is the web server attempting to execute user created CGI scripts.

existing policy.

The feasibility of our approach is demonstrated by *monkey.org*, a Unix shell provider in Ann Arbor, who uses Systrace to sandbox over two hundred users. They generated separate policies for approximately 250 applications.

An analysis of the policies shows that applications are allowed to call seventy one different system calls on average; see Figure 4. Usually Unix systems support several hundred system calls. When an adversary gains control over an application, she may attempt to obtain higher privileges by using all possible system calls[3]. By limiting the adversary to only those system calls required by the application, we reduce her potential to cause damage.

We notice two peaks, one at sixty four system calls and the other one at eighty seven. The first peak is caused by policies for standard Unix utilities like *chmod*, *cat*, *rmdir* or *diff* all of which have similar policies. The second peak is caused by identical policies for the different utilities in the MH message system, which require more system calls for establishing network connections and creating files in the filesystem.

Most of the policy statements specify access to the filesystem: 24% of them control *read* access, 6% *write* access and 5% the execution of other programs.

[3]Recently discovered vulnerabilities in Unix operating systems allow an adversary to execute code in kernel context due to incorrect argument checking on system calls [9, 29].

6.3 Intrusion Detection and Prevention

The capability for intrusion detection and prevention follows automatically from our design. System calls that violate the policy are denied and recorded by the operating system. This prevents an adversary from causing damage and creates an alert that contains the restricted operation.

A correct policy restricts an application to only those operations required for its intended functionality. While this prevents an adversary from harming the operating system arbitrarily, she may still abuse an application's innate functionality to cause damage. We employ audit trails to log potentially malicious activity not prevented by policy.

At *monkey.org*, Systrace generated approximately 350,000 log entries for 142 users over a time period of two months. The system is configured to log all denied system calls as well as calls to *execve* and *connect*. By correlating the number of programs executed with the number of policy violations for all users, we identify those users that exhibit unusual behavior. In Figure 5, we notice a few users that generate an unproportionally high number of policy violations compared to the number of programs they execute. In this case, the user with the most policy violations is the web server attempting to execute user created CGI scripts. The user that executes the most applications without frequent policy violations uses MH to read her email.

Mode	Real time in μsec	User time in μsec	System time in μsec
Normal	0.35 ± 0.00	0.14 ± 0.03	0.22 ± 0.03
In-kernel	0.46 ± 0.01	0.17 ± 0.04	0.28 ± 0.04
User space	37.71 ± 0.18	0.30 ± 0.07	5.60 ± 0.61

Figure 6: A microbenchmark to compare the overhead of a single *geteuid* system call for an unmonitored process and for process confinement with different policies. Making a policy decision in the kernel is considerably faster than requesting a policy decision from the user space policy daemon.

6.4 Limitations

Although powerful, policy enforcement at the system call level has inherent limitations. Monitoring the sequence of system calls does not give complete information about an application's internal state. For example, system services may change the privilege of a process on successful authentication but deny extra privilege if authentication fails. A sandboxing tool at the system call level cannot account for such state changes. However, it is still possible to enforce global restrictions that state, for example, that *root* should never be allowed to login. This is possible because those restrictions do not depend on an application's internal state.

To increase the security of authentication services like SSH, it is possible to use a combination of *privilege separation* [30] and *system call policy enforcement.* With privilege separation, the majority of an application is executed in an unprivileged process context. Vulnerability in the unprivileged code path should not lead to privilege escalation. However, in a Unix system an unprivileged process can still execute system calls that allow local network access. Using Systrace to sandbox the application, we can prevent the unprivileged process from executing any system calls that are not necessary for its functionality.

7 Performance

To determine the performance impact of Systrace, we measured its overhead on the execution time of single system calls and on several applications. All measurements were repeated at least five times on a 1.14 GHz Pentium III running OpenBSD. The results are displayed as averages with corresponding standard deviation.

We conduct the microbenchmarks of a single system call by repeating the system call several hundred thousand times and measuring the real, system, and user time. The execution time of the system call is the time average for a single iteration.

Mode	Real time in μsec	User time in μsec	System time in μsec
Normal	5.52 ± 0.01	0.34 ± 0.20	5.08 ± 0.16
In-kernel	5.88 ± 0.03	0.31 ± 0.22	5.55 ± 0.22
1-deep	139.20 ± 0.09	0.56 ± 0.12	15.80 ± 1.01
2-deep	167.72 ± 0.41	0.64 ± 0.18	15.84 ± 1.10
3-deep	198.34 ± 0.67	0.40 ± 0.17	18.28 ± 0.38
4-deep	231.121 ± 0.27	0.43 ± 0.13	19.40 ± 1.39

Figure 7: A microbenchmark to compare the overhead of the *open* system call. Due to filename normalization, the time to make a policy decision in user space depends on the number of components in the filename. Every component adds about 30 μsec.

As a baseline, we measure the time for a single *geteuid* system call without monitoring the application. We compare the result with execution times obtained by running the application under Systrace with two different policies. The first policy permits the *geteuid* via the in-kernel policy table. For the second policy, the kernel consults the user space policy daemon for a decision. We see that the *in-kernel* policy evaluation increases the execution time by $31\% \pm 3\%$ and that slightly more time is spent in the kernel. When the kernel has to ask the user space daemon for a policy decision, executing a single system call takes much longer, mostly due to two context switches required for every policy decision. The results are shown in Figure 6.

The *open* system call requires more work in the kernel than *getuid*. A microbenchmark shows that the in-kernel evaluation of the policy increases the execution time by $7\% \pm 0.6\%$. The execution time for a user space policy decision depends on the depth of the file in the directory tree. When the path to the filename has only one component, the increase in execution time is over 25-fold. Each directory component in the path adds approximately thirty microseconds to the execution time due to filename normalization, as shown in Figure 7.

The last microbenchmark measures the overhead of using the *read* system call to read a 1 kbyte buffer from `/dev/arandom`, which outputs random data created by a fast stream cipher. There is no noticeable difference in execution time and system time increases by less than 1% for in-kernel policy evaluation. We omit measurement of user space because *read* requires no user

Mode	Real time in μsec	User time in μsec	System time in μsec
Normal	37.61 ± 0.03	0.11 ± 0.11	37.34 ± 0.10
In-kernel	37.61 ± 0.03	0.14 ± 0.16	37.45 ± 0.21

Figure 8: A microbenchmark to compare the overhead of the *read* system call when reading a 1 kbyte buffer from /dev/arandom. In this case, there is no measurable performance penality for the in-kernel policy decision.

File size in MByte	Normal	Systrace	Increase in percent
0.5	0.88 ± 0.04	0.92 ± 0.07	4.5 ± 9.3
1.4	2.51 ± 0.01	2.52 ± 0.01	0.4 ± 0.6
2.3	4.15 ± 0.01	4.17 ± 0.01	0.5 ± 0.3
3.2	5.62 ± 0.01	5.64 ± 0.01	0.4 ± 0.3
4.0	7.18 ± 0.03	7.18 ± 0.03	0.0 ± 0.6
4.9	8.55 ± 0.01	8.57 ± 0.02	0.2 ± 0.3

Figure 9: A macrobenchmark comparing the runtime of an unmonitored gzip process to gzip running under Systrace. Because this benchmark is computationally intensive, policy enforcement does not add a significant overhead.

space policy decision. The results are shown in Figure 8.

Enforcing system call policies adds overhead to an application's execution time, but the overall increase is small, on average.

Figure 9 compares the runtime of *gzip* for different file sizes from 500 kByte to 5 MByte. Gzip executes thirty system calls per second on average, most of them *read* and *write*. In this case, the execution time is not significantly effected by Systrace, because the application spends most of its time computing, and executes relatively few system calls.

To assess the performance penality for applications that frequently access the filesystem, we created a benchmark similar to the Andrew benchmark [22]. It consists of copying a tar archive of the Systrace sources, untarring it, running configure, compiling the sources and then deleting all files in the source code subdirectory.

During the benchmark, forty four application programs are executed. We use Systrace to generate policies automatically, then improve the policies that result with a simple script. The benchmark executes approximately 137,000 system calls. A decomposition of the most frequent system calls is shown in Figure 10. The system call with the highest frequency is *break* which is used to allocate memory. System calls that access the

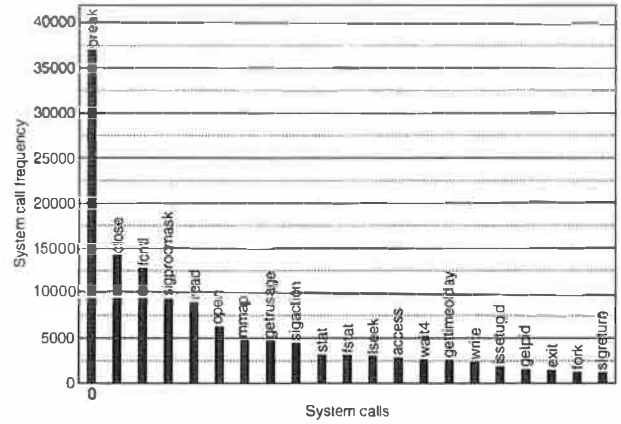


Figure 10: Histogram of system call frequency for compilation benchmark. The performance impact of application confinement depends mostly on the number of system calls that require a policy decision by the user space daemon. The histogram shows that the most frequent system calls can be handled by the in-kernel policy.

Benchmark	Normal in sec	Systrace in sec	Increase in percent
Compile	10.44 ± 0.09	13.71 ± 0.09	31 ± 1.4
Crawler	0.84 ± 0.03	0.88 ± 0.03	4.8 ± 5.2
Gzip-4.9M	8.55 ± 0.01	8.57 ± 0.02	0.2 ± 0.3

Figure 11: Overview of different macrobenchmarks comparing the execution time of an unmonitored run with the execution time running under Systrace. The compilation benchmark incurs the highest performance penality. On the other hand, it is very complex, consisting of more than forty applications and still shows acceptable performance. Running the other benchmarks with Systrace incurs only small performance penalties.

filesystem are also prominent.

A direct comparison between the execution times is shown in Figure 11. Under Systrace, we notice an increase in running time by $31\% \pm 1.4\%$. The number of executed system calls increases to approximately 726,000 because filename normalization requires the *getcwd* function, which causes frequent calls to *lstat* and *fstat*. Running the same benchmark under NetBSD 1.6I shows a significantly smaller increase in system calls because it implements a *getcwd* system call.

A second macrobenchmark measures the runtime of a web crawler that downloads files from a local web server. The crawler retrieves approximately one hundred forty files with an average throughput of two megabytes per second. For this macrobenchmark,

the running time under Systrace increases only by $4.8\% \pm 5.2\%$; see Figure 11.

The additional cost of Systrace, although noticeable is not prohibitive, especially for interactive applications like web browsers, in which there is no observable performance decrease for the end user.

8 Future Work

This work opens up many avenues for future research. Systrace may be used for quality assurance by injecting random faults into a running application. This allows us to introduce error conditions that are not normally triggered and to observe if the application recovers correctly from them. For example, we may simulate resource starvation such as a full filesystem or out-of-memory conditions. Using argument replacement, it is possible to change the way an application interacts with the operating system. By changing filename arguments, it is possible to present a virtual filesystem layout to the application. We may also rewrite the addresses an application attempts to access on the network. This allows us to redirect network traffic to different hosts or to application-level firewalls.

9 Conclusion

This paper presented a new approach for application confinement that supports automatic and interactive policy generation, auditing, intrusion detection and privilege elevation and applies to both system services and user applications. We argued that system call interception is a flexible and appropriate mechanism for intrusion prevention. Our hybrid implementation enables fail-safe operation while maintaining low performance overhead and good portability. This paper addressed important issues not addressed by previous research. The translation of system call arguments into human-readable strings allows us to design a simple policy language. It also enables our system to generate fine grained policies both automatically and interactively. The resulting policies restrict applications without breaking their functionality.

Privilege elevation in conjunction with application confinement allows us to reduce significantly the privileges required by system services. Using privilege elevation, we assign fine-grained privileges to applications without requiring the *root* user. Instead of retaining *root* privileges throughout an application's lifetime, an application may run without special privileges and receive elevated privileges as determined by policy.

Our security analysis discussed how we overcome problems common to system call interception tools and how our design allows for further functionality such as intrusion detection and prevention.

We analyzed the performance of Systrace and showed that additional performance overhead is acceptable and often not observable by the user of a sandboxed application.

10 Acknowledgments

I would like to thank Peter Honeyman, Terrence Kelly, Chuck Lever, Ken MacInnis, Joe McClain, Perry Metzger and Jose Nazario for careful reviews. I also thank Marius Eriksen, Angelos Keromytis, Patrick McDaniel, Perry Metzger, Dug Song and Markus Watts for helpful discussions on this topic.

References

[1] Anurag Acharya and Mandar Raje. MAPbox: Using Parameterized Behavior Classes to Confine Applications. In *Proceedings of the 9th USENIX Security Symposium*, August 2000. 2, 4, 9

[2] Albert Alexandrov, Paul Kmiec, and Klaus Schauser. Consh: Confined Execution Environment for Internet Computations, 1998. 2

[3] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, and Sheila A. Haghighat. A Domain and Type Enforcement UNIX Prototype. In *Proceedings of the 5th USENIX Security Symposium*, pages 127–140, June 1995. 2

[4] Steven M. Bellovin. Computer Security - An End State? *Communications of the ACM*, 44(3), March 2001. 1

[5] Matt Bishop. How to write a setuid program. *;login;*, 12(1):5–11, 1987. 1, 3

[6] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos Keromytis. The KeyNote trust-management system version 2. RFC 2704, September 1999. 4

[7] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized Trust Management. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 164–173, May 1996. 4

[8] CERT. OpenBSD contains buffer overflow in "select" call. Vulnerability Note VU#259787, August 2002. `http://www.kb.cert.org/vuls/id/259787`. 1

[9] Silvio Cesare. FreeBSD Security Advisory FreeBSD-SA-02:38.signed-error. `http://archives.neohapsis.com/archives/freebsd/2002-08/0094.html`, August 2002. 10

[10] Suresh N. Chari and Pau-Chen Cheng. BlueBox: A Policy-driven, Host-Based Intrusion Detection System. In *Proceedings of the ISOC Symposium on Network and Distributed System Security*, Feburary 2002. 2

[11] Hao Chen, David Wagner, and Drew Dean. Setuid Demystified. In *Proceedings of the 11th Usenix Security Symposium*, August 2002. 1

[12] William R. Cheswick and Steven M. Bellovin. *Firewalls and Internet Security Repelling the Willy Hacker.* Addison-Wesley Publishing Company, 1994. 1

[13] M. Coleman. Subterfogue: A Framework for Oberserving and Playing with Reality of Software. `http://subterfugue.org/`. 2

[14] Pawl J. Dawidek. Cerb: System Firewall Mechanism. `http://cerber.sourceforge.net/`. 2

[15] G. Fernandez and L. Allen. Extending the UNIX Protection Model with Access Control Lists. In *Proceedings of the Summer 1988 USENIX Conference*, pages 119–132, 1988. 1, 2

[16] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A Sense of Self for Unix Processes. In *Proceedings of the 1996 IEEE Symposium on Research in Security and Privacy*, pages 120–128, 1996. 2, 5

[17] Przemyslaw Frasunek. ntpd $\leq$ 4.0.99k remote buffer overflow. Bugtraq, April 2001. CVE-2001-0414. 6

[18] Tal Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Proceedings of the ISOC Symposium on Network and Distributed System Security*, 2003. 2, 8

[19] Douglas P. Ghormley, Steven H. Rodrigues, David Petrou, and Thomas E. Anderson. SLIC: An Extensibility System for Commodity Operating Systems. In *Proceedings of the USENIX 1998 Annual Technical Conference*, pages 39–52, June 1998. 3

[20] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A Secure Environment for Untrusted Helper Applications. In *Proceedings of the 6th Usenix Security Symposium*, July 1996. 1, 2, 4

[21] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion Detection Using Sequences of System Calls. *Journal of Computer Security*, 6(3):151–180, 1998. 1, 2, 5, 9

[22] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988. 12

[23] Sotiris Ioannidis, Steven M. Bellovin, and Jonathan M. Smith. Sub-Operating Systems: A New Approach to Application Security. In *Proceedings of the SIGOPS European Workshop*, September 2002. 2

[24] K. Jain and R. Sekar. User-Level Infrastructure for System Call Interposition: A Platform for Intrustion Detection and Confinement. In *Proceedings of the ISOC Symposium on Network and Distributed System Security*, February 2000. 1, 2, 4

[25] Calvin Ko, George Fink, and Karl Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings of the 10th Annual Computer Security Applications Conference*, pages 134–144, December 1994. 3

[26] Henry M. Levy. *Capability-Based Computer Systems.* Digital Press, 1984. `http://www.cs.washington.edu/homes/levy/capabook/`. 2

[27] Vern Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *Proceedings of the 7th USENIX Security Symposium*, January 1998. 1

[28] David S. Peterson, Matt Bishop, and Raju Pandey. A Flexible Containment Mechanism for Executing Untrusted Code. In *Proceedings of the 11th USENIX Security Symposium*, pages 207–225, August 2002. 2

[29] Niels Provos. OpenBSD Security Advisory: Select Boundary Condition. `http://monkey.org/openbsd/archive/misc/0208/msg00482.html`, August 2002. 10

[30] Niels Provos. Preventing Privilege Escalation. Technical Report CITI 02-2, University of Michigan, August 2002. 1, 11

[31] Thomas Ptacek and Timothy Newsham. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Secure Networks Whitepaper, August 1998. 1

[32] Jerome H. Saltzer. Protection and the Control of Information in Multics. *Communications of the ACM*, 17(7):388–402, July 1974. 1, 2

[33] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. In *Proceedings of the IEEE 69*, number 9, pages 1278–1308, September 1975. 6

[34] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, David Andersen, and Jay Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. In *Proceedings of the 8th Usenix Security Symposium*, pages 123–139, August 1999. 2, 8

[35] David Wagner and Drew Dean. Intrusion Detection via Static Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2001. 3, 9

[36] David Wagner and Paolo Soto. Mimicry Attacks on Host-Based Intrusion Detection Systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, November 2002. 1, 3, 5, 9

[37] David A. Wagner. Janus: an Approach for Confinement of Untrusted Applications. Technical Report CSD-99-1056, 12, 1999. 2, 8

[38] Kenneth M. Walker, Daniel F. Sterne, M. Lee Badger, Michael J. Petkac, David L. Shermann, and Karen A. Oostendorp. Confining Root Programs with Domain and Type Enforcement (DTE). In *Proceedings of the 6th Usenix Security Symposium*, July 1996. 2

[39] Andreas Wespi, Marc Dacier, and Hervé Debar. An Intrusion-Detection System Based on the Teiresias Pattern-Discovery Algorithm. In *Proceedings of the EICAR*, 1999. 3

Scrash: A System for Generating Secure Crash Information

Pete Broadwell Matt Harren Naveen Sastry*
University of California, Berkeley
{pbwell, matth, nks}@cs.berkeley.edu

Abstract

This paper presents Scrash, a system that safeguards user privacy by removing sensitive data from crash reports that are sent to developers after program failures. Remote crash reporting, while of great help to the developer, risks the user's privacy because crash reports may contain sensitive user information such as passwords and credit card numbers. Scrash modifies the source code of C programs to ensure that sensitive data does not appear in a crash report. Scrash adds only a small amount of run-time overhead and requires minimal involvement on the part of the developer.

1 Introduction

Developers often examine a failed program's state to diagnose and fix software bugs. For this reason, operating systems and programming suites include tools to capture a program's state in a core file at crash time. The recent advent of ubiquitous network connectivity for personal computers makes *remote crash reporting* possible, whereby programs send crash information back to developers after a failure. This practice, which allows developers to receive information about bugs in their programs after the programs have been distributed to users, has since become commonplace [1, 2, 3, 4]. Remote crash reporting offers many readily apparent benefits to developers, but the privacy-related implications of the technology are not as well understood.

The contributions of this paper include

- An explanation of the privacy-related problems posed by remote crash reporting.
- An analysis of the tradeoffs inherent in the design of core file cleaning systems.
- A description of an easy-to-use system that safeguards the user's private information from exposure via crash reports.

*The authors were supported in part by DARPA NEST contract F33615-01-C-1895, NSF grants CCR-0093337, CCR-0085899, CCR-0085949, and CCR-0081588, and a grant from Microsoft.

1.1 Crash reporting background

Remote crash reporting tools appear in many forms, but all perform the same basic task: gathering and transmitting crash-related data to a remote database. Microsoft's Dr. Watson tool for Windows [4] performs crash-reporting services for most Microsoft applications. Another Windows-based tool is BugToaster [2], a third-party crash data collection utility that sends its reports to an independent database. Bug-Buddy for the GNOME desktop environment [1] and Mozilla Talkback [3] are remote crash reporting tools used in the open source community.

The types of data contained in a remote crash report can vary widely, depending upon the configuration of the reporting tool. Crash reporting tools typically record some information about the environment in which the failed program was running, including the error associated with the crash, program version information, loaded drivers, memory usage and open files. They also send back a subset of the data present in a core file: part or all of the call stack, processor registers, and heap contents of the crashed program.

Remote crash reporting technology grants the developer access to potentially vast amounts of crash data, speeding the diagnosis and repair of software vulnerabilities. For example, developers can fingerprint the call stacks returned in crash reports to determine which bugs appear most often and thus deserve the most attention. The developer also can suggest fixes or patches to the user based on the contents of a crash report.

Given the increasing prevalence of remote crash reporting, it is important to consider the security-related risks associated with the technology. Because they contain some or all of the memory contents of the program at the time it failed, crash reports may include sensitive user information such as credit card numbers, passwords or web browser cookies. A recent Department of Energy security advisory regarding the Dr. Watson crash report tool for Office XP and Internet Explorer warns that the program could send sensitive information to Microsoft, since the memory dump in the crash report might contain portions of the document being viewed [12]. A related concern about Dr. Watson is that the program stores comprehensive crash reports in a world-readable directory on the host computer [18]. This practice

raises security and privacy concerns, because a malicious party on a multi-user system could examine the crash report and extract confidential information.

There are inherent privacy risks associated with sending crash data to a remote party over a network. An initial vulnerability is that the data may be intercepted en route. Dr. Watson guards against this threat by encrypting the data stream with SSL [4], but GNOME's Bug-Buddy currently sends crash reports unencrypted via `sendmail` [1].

The primary concern, though, is the fate of the failure data after it reaches the crash data repository. These repositories contain crash reports from many users, and may become popular targets if they are known to house sensitive information. An important distinction here is that the user trusts the developer to produce quality software – the user installs and executes the software voluntarily, after all. The user should not, however, be obligated to trust the developer to safeguard his sensitive crash data for an indefinite length of time. Securely maintaining data takes a different kind of expertise than writing secure and correct code. Thus, users may be more willing to participate in remote crash reporting if the crash reports can be stripped of personal information.

We also hypothesize that developers often won't want to store a user's sensitive information. The inclusion of privacy-sensitive information in the crash report presents a risk for the developer: a security breach of a crash repository could result in bad publicity or financial liability. For these reasons, we believe that *both* users and developers would like to eliminate sensitive information from crash data.

2 Core File Filtering Systems

A *core file* is a snapshot of a program's execution state generated when a crash occurs. We propose a core file *filtering system* as a method of identifying *sensitive information* and ensuring that it does not appear in a core file. The developer decides which categories of data should be considered "sensitive" for each particular executable; the filtering system must then prevent sensitive information from appearing in the final crash report. Conversely, *insensitive* information is allowed to appear in the crash report. A filtering system is composed of two separate phases: the first phase transforms the application source code, and the second phase transforms core files that result from application crashes.

We place two restrictions on the source code modification phase: the behavior of the application to be modified must be indistinguishable from that of the original, and the transformation should not modify the program in a way that makes debugging the filtered core file unnecessarily difficult. Since the filtering system is supposed to preserve a developer's ability to debug the original application, the transformation must preserve the variables and control structure of the application to the greatest extent possible. For example, we allow transformations that move the memory locations of variables since the contents of these variables are still present in the resulting core file. Thus, an *information-preserving* source code transformation retains all of the same variables of the original program but may rearrange their layout in memory.

The second phase of a filtering system modifies the core file generation process so that no sensitive data appears in the core file. In practice, this task can be accomplished by running a separate program to delete selected information from a complete core file after it has been generated.

We now outline two metrics to characterize the effectiveness of the filtering system. The first metric measures the usefulness of the core file to the developer, since debugging a crash is more difficult if a critical piece of data has been removed from the core file. Using this metric, the original, full core file is the most useful for debugging, while an empty core file is useless. The second metric measures the filtering system's effectiveness from the user's perspective, i.e., how well the system protects the user's privacy and data. Using this metric, a user's privacy is best preserved if the filter removes all information.

The challenge, then, of designing a filtering system involves balancing the needs of the developer with those of the user. The filtering system must preserve as much information as possible for the developer while maintaining privacy for the user. A developer may choose any number of different privacy guarantees, depending on the particular application and the degree to which privacy is necessary. One such guarantee, for example, may prevent passwords from being leaked, but may not conceal the length of the password if this value is useful for debugging.

This model assumes that the developer is trustworthy. It does not guard against privacy violations by malicious developers, since a developer can easily insert a covert channel into the program. Rather, the developer controls the filtering system and defines the balance between the user's privacy and the developer's need to debug the application. We imagine that advanced filtering systems might even give the user a choice between multiple privacy-utility tradeoffs. Thus, the primary goal of a filtering system is to protect against privacy violations after the core file has been generated, particularly in crash repositories.

2.1 Scrash goals

Our system, Scrash, is an easy-to-use filtering system that presents several tradeoffs between privacy guarantees and developer utility of crash data. Its goal is to eliminate sensitive memory locations and their copies from a core file. In addition, Scrash provides developer control over certain classes of derivative data that may be removed from the core file. For example, Scrash considers the length of a sensitive

buffer to be sensitive as well, which ensures that the length of a sensitive password buffer computed via `strlen` will also be regarded as sensitive. The developer may choose to override this rule, however, if she feels that disclosing the length of the buffer may be beneficial for problem debugging and does not pose a significant privacy risk.

Scrash ignores privacy leaks resulting from indirect information flows or other covert channels. As an example of such an information flow technique, the program counter and call stack can leak information on the state of sensitive variables. Consider the following example:

```
char c = password[0];
if (c >= 'a' && c <= 'z') {
  // stmt a
} else {
  // stmt b
}
```

If the program's execution state indicates that statement *b* was executed, then an adversary can infer that the password does not start with a lower case letter even if the password variable is marked as sensitive. Eliminating control flow privacy leaks and other covert channels while retaining enough information for debugging is difficult, so Scrash ignores such vulnerabilities. For example, the processor registers and even the entire call stack would not be available to the developer in a system that seeks to guard against control flow privacy leaks. All reveal the state of prior control flow decisions and could be used to discover information about the state of sensitive variables that had been used in conditionals.

3 Implementation

Scrash seeks to eliminate sensitive information from the heap, stack, and global variables while still providing useful information to the developer. We perform source code transformations to place the contents of any sensitive variables in a separate region of memory, which we then erase during core file generation to ensure that it is not transmitted as part of a crash report. Thus, the stack, globals and main heap in our modified core file will only contain insensitive information, so that the crash reporting tool is free to transmit any of these regions. The key difficulty of this task, which we will address below, is identifying the sensitive data. Even though the heap is not often transmitted using current crash reporting tools, we make a distinction between the sensitive and insensitive heap in the case that it may be transferred when sending a more detailed crash report. Making this distinction has a negligible performance cost, so we view the added safety it provides as worthwhile.

We implemented the source code transformation phase in 1200 lines of new Objective Caml code. We link the modified application with a memory allocator to which we added 250 lines of new C code. We wrote the cleaning phase using 90 lines of C code.

3.1 Merging of source files

We use CIL (a C Intermediate Language implemented in OCaml) [11] as the infrastructure for our source-to-source translation. CIL translates C code into a clean, easy-to-manipulate subset of C. It includes drivers that act as drop-in replacements for `gcc`, `ar`, and `ld` so that CIL can be used with existing makefiles. CIL uses these drivers to collect all of the source files for a program, preprocess them, and merge them into a single C file to facilitate whole-program analysis.

3.2 Analyzing the sensitivity of variables

Our system extends each type in a C program with a *type qualifier* to indicate whether or not it may hold sensitive information. Type qualifiers are an additional specification of the traditional C types. For example, "`$sensitive int`" is the type of an integer variable that may hold sensitive information at some point during its lifetime. When declaring a variable, the developer can specify that the variable will contain sensitive information by adding the `$sensitive` annotation. For all unannotated variables, we use the CQual type qualifier inference engine to determine whether the variable may hold sensitive information [14].

CQual performs an interprocedural program analysis to determine where sensitive data might flow from the initial set of sensitive variables annotated by the programmer. If CQual detects an assignment from a sensitive variable to an "unconstrained" variable, the unconstrained variable will be considered sensitive. Thus, CQual determines where the `$sensitive` qualifier spreads throughout the program. After CQual has finished, we know that all remaining unconstrained variables only contain insensitive data, since they never receive any assignments from sensitive variables. Conversely, if CQual determines that a variable is sensitive, it may contain sensitive information during the execution of the program, since there is a possible assignment to it from a known sensitive variable. The question of whether data may be sensitive is analogous to the question of whether it may be *tainted*, so we can use the same analysis as in Shankar et al. [14].

As an alternative to annotating specific data at the point it enters the program, the programmer may choose to use a pre-annotated header file that marks as sensitive all data returned by functions like `read` and `recv`. At the cost of unnecessarily marking some values as sensitive, this option makes it easy to denote user data as sensitive without the

need to enter program-specific annotations. We take this approach in our evaluation experiments.

The CQual stage outputs the original program with attributes added to each variable describing its sensitivity. These annotations allow later stages of Scrash to determine whether a variable should reside in the secure or insecure region of memory.

3.3 Smalloc: secure malloc

After identifying sensitive variables, it becomes possible to erase their contents before shipping the core file. A difficulty arises in determining where the information resides in the core file, however, since in general the sensitive variables will be scattered throughout the entire core file. We need a way to communicate sensitivity information from the static analysis to the runtime cleaning process.

One method to recognize sensitive variables would be to append an immutable tag to each sensitive variable; the tag would describe the variable's sensitivity status. The post-processing cleaning step could then iterate over the core file and remove or overwrite all sensitive variables by checking for the tag.

An alternative approach, which we utilize, groups sensitive memory locations together and places an identifying header at the start of the region. This approach is ultimately more space-efficient than tagging each variable separately and simplifies the process of removing sensitive data from the core file.

We have written Smalloc, a region-aware memory allocator, to manage this "secure" region. It is based on the Vmalloc package, which provides an ideal platform for creating allocators [15]. The interface to Smalloc is similar to `malloc`. The only difference is that we add an extra parameter to the `smalloc` function to identify whether the new memory should be allocated in the sensitive or insensitive memory region. The signatures of the `realloc` and `free` functions remain unchanged. See Figure 2 for the complete Smalloc interface.

Smalloc creates sensitive memory regions for heap allocated variables, sensitive global variables, and the sensitive stack. We will discuss these regions below. Each of the regions is actually embedded within the normal heap segment. The globals and stack regions are statically sized and allocated at program initialization. The size of the sensitive heap region is dynamic.

3.4 Transformations

We perform transformations on the program source code to ensure that the variables the CQual phase marks as sensitive are placed into the sensitive memory region by Smalloc library routines. CIL provides an ideal platform for performing these transformations. We outline each of the transfor-

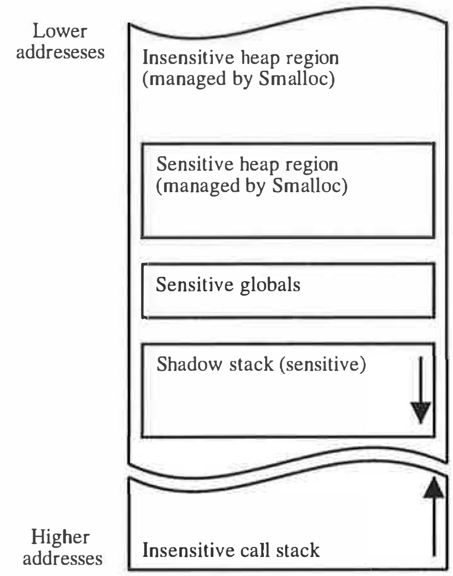


Figure 1: Layout of a process's memory when using Scrash. Both heap regions are managed by Smalloc. The sensitive globals, sensitive stack, and sensitive heap are embedded within the insensitive heap region. The arrows indicate the direction of stack growth.

```
void * smalloc (size_t size, int issecure);
void * scmalloc (size_t nmemb, size_t size, int issecure);
void sfree (void * ptr);
void * srealloc (void * ptr, size_t size);
```

Figure 2: The Smalloc allocator interface. The allocation functions take an extra boolean parameter that specifies whether the data should be allocated in the sensitive region or on the insecure heap.

```
#include <crypt.h>
#include <malloc.h>
#include <string.h>
int $sensitive private[2] = {0, 1};

void getPassword(char cryptpw[14]) {
  char $sensitive * password = malloc (255);
  memcpy (cryptpw, crypt (password, "00"), 14);
}

void check() {
  char $sensitive cryptpw[14];
  getPassword(cryptpw);
}
```

Figure 3: A sample code fragment that we will use to illustrate some of the transformations that Scrash uses (see Figure 4). It contains a sensitive global, a pointer to sensitive data, and a sensitive stack variable.

```
typedef unsigned int size_t;
struct check_shadow {
   char cryptpw[14] ;
};
struct __smalloc_globals {
   int private[2] ;
};
struct __smalloc_globals *__smalloc_global_var ;
void ( __attribute__((__constructor__)) __smalloc_global_init)() ;
char *stackPointer    =    0;
void *srealloc(void *ptr , unsigned int size ) ;
void sfree(void *ptr ) ;
void *scalloc(unsigned int nmemb, unsigned int size, int issecure ) ;
void *smalloc(unsigned int size , int issecure ) ;
extern char *crypt(char const *__key , char const *__salt ) ;
extern void *malloc(size_t __size ) ;
extern void *memcpy(void * __restrict __dest ,
                    void const *__restrict __src, size_t __n) ;
void getPassword(char *cryptpw ) {
  char *password ;
  char *tmp ;
  void const    * __restrict     tmp___0 ;
  {
    tmp = (char *)smalloc(255U, 1);
    password = tmp;
    tmp___0 = (void const *)
      crypt((char const *)password, (char const*)"00");
    memcpy((void *)cryptpw, tmp___0, 14U);
    return;
  }
}
void check(void) {
  struct check_shadow *check_shadow ;
  {
    check_shadow = (struct check_shadow *)stackPointer;
    stackPointer = stackPointer + sizeof(struct check_shadow );
    getPassword((char *)(check_shadow->cryptpw));
    {
      stackPointer = (char *)check_shadow;
      return;
    }
  }
}
void __smalloc_global_init(void) {
  {
    __smalloc_global_var = (struct __smalloc_globals *)
        smalloc(sizeof(struct __smalloc_globals ), 1);
    __smalloc_global_var->private[0] = (int )0;
    __smalloc_global_var->private[1] = (int )1;
  }
}
```

Figure 4: The results of running Scrash on the code fragment from Figure 3. Note that the constructor function `__smalloc_global_init` allocates the sensitive global inside the `__smalloc_globals` structure, which is allocated on the sensitive heap. This constructor function runs prior to `main` and is specified with the `__constructor__` attribute. The sensitive heap variable `password` is now allocated on the secure heap. Finally, the sensitive local array `cryptpw` is allocated on the shadow stack. A new structure, `check_shadow`, contains this variable. Maintenance of the shadow stack is performed on entry and exit of the `check` function.

mations below. The results of applying the complete set of transformations to the program in Figure 3 can be seen in Figure 4.

3.4.1 Sensitive heap variables

We allocate memory on the sensitive heap when the results of a `malloc` call are assigned to a pointer declared with the `$sensitive` qualifier. Recall that CQual assigns this qualifier to variables that could potentially contain sensitive information. Similarly, the absence of the `$sensitive` qualifier on a pointer indicates that the memory should be allocated on the insensitive heap. Thus, we change each of the allocation calls to use the Smalloc allocator, using the presence of the `$sensitive` attribute to denote which region the Smalloc allocator uses. We similarly replace `calloc` with `scalloc`.

In addition to replacing the allocation functions, we replace any instances of `free` and `realloc` with the Smalloc equivalents: `sfree` and `srealloc`. These functions have the same arguments and return types as the functions they replace, so we can perform a simple substitution.

3.4.2 Sensitive stack variables

There are two possible transformations that can be applied to place sensitive stack variables within the secure memory region:

Heap allocation of local variables. This transformation moves the sensitive stack variables into the secure heap. At function entry, we allocate a block of space on the secure heap for all of the sensitive local variables, which we deallocate before exiting. We also rewrite all references within the function to point to the reallocated stack variables.

This transformation, however, requires adding a `smalloc` and `sfree` call to any function with sensitive stack variables. We found that these extra calls had a significant impact on performance (see Section 4.2), so we developed an alternative transformation for stack variables:

Shadow stack. A shadow stack is a separate area of memory that parallels the normal stack and holds sensitive variables. The shadow stack resides within the secure region, maintaining the invariant that all sensitive information is contained within that region. The shadow stack's size is set to the maximum size of the program's regular stack. We insert code to adjust the shadow stack pointer, which we implement as a global variable, at the entry and exit points of each function. This approach offers better performance than allocating all local variables on the secure heap.

Every time control reaches a function body entry point, the shadow stack pointer is incremented by the combined size of all of the sensitive variables for that frame. Thus, the shadow stack grows toward higher memory addresses.

We rewrite all accesses to variables declared with the `$sensitive` qualifier to use the new sensitive stack. We also insert code to decrement the stack pointer just before control leaves the end of the function body. After exiting a function, the memory located at higher addresses than the current shadow stack pointer is unused, but it still contains the remnants of the sensitive information that the function body placed there. We could overwrite the contents of this memory to eliminate the leftover values, but since the shadow stack is allocated within the sensitive region, it will be overwritten during the core file cleaning process anyway. Thus, overwriting the unused portion of the shadow stack is an unnecessary step, as the cleaning process will erase all of the stack contents, even the unused portions. See Section 3.5 for a description of the cleaning process.

3.4.3 Sensitive global variables

Finally, we define a new structure to contain all of the sensitive global variables, instantiating it as `__smalloc_global_var`. We allocate this structure on the secure heap with a special initialization function, using the `gcc`-specific attribute "constructor" to ensure that this function runs before `main()`. We also perform any initializations that are needed for sensitive global variables by expanding their initializer clauses into regular C statements and placing them in the constructor function.

3.5 Postprocessing: cleaning the core file

After employing the above transformations, all of the program's sensitive information will be fully contained within the secure memory region. The core file will still contain the sensitive information, however, if the program crashes and no further filtering steps are taken. At this point, we use a cleaning process to overwrite the secure region of a core file after it has been generated. The cleaner operates by first searching for a special tag that identifies the metadata for the secure region. The metadata encodes the type and size of the region, allowing the cleaning process to overwrite it.

Recall that the secure heap region is dynamically sized. When the region changes size, its new size is reflected in the metadata. If the region shrinks, memory that previously contained sensitive data will remain *outside* of the sensitive region. Thus, the cleaning process will not overwrite it. To maintain the invariant that sensitive data resides only within sensitive regions, Smalloc overwrites the contracted memory whenever a sensitive region shrinks.

The cleaning process must take special care to ensure that an adversary does not trick the cleaner into leaving portions of the sensitive region intact. Consider a cleaning process that scans through the core file sequentially, searching for the metadata that marks the boundary and size of a secure region and then erasing the specified number of bytes after the tag. A crafty adversary could arrange for a counterfeit secure region tag to appear in an insensitive memory region prior to the secure region, and construct the metadata so that the cleaner overwrites the actual secure region tag. Since the real secure region tag has at this point been erased from the core file, the cleaner would find no further tags and go on to generate a core file that still contains sensitive information.

To counter this attack, we wrote the cleaner to locate all secure region tags that might appear within a core file first, and then perform the overwriting. This approach prevents a metadata entry earlier in the core file from causing the cleaner to disregard a later one. Thus, all sensitive information will still be removed from the core file. An attack of this form may still induce the cleaner to remove insensitive data from the core file, but this is only a denial of service attack. The shortcoming doesn't represent a privacy or security threat, though it could hinder the developer from debugging the core file. We view guarding against denial of service attacks as a secondary concern, compared to protecting the user's privacy.

One could imagine incorporating core file cleaning into the operating system routines that produce the core file. Making this change would ensure that the cleaning process always runs before the crash report is written to disk, and would prevent problems such as the Dr. Watson bug mentioned in the introduction.

3.6 Implementation details

3.6.1 Threads

The Vmalloc package, on which the Smalloc allocator is built, is thread-safe, so extending our design to multithreaded programs is straightforward. If sensitive local variables are transformed into heap-allocated structures, no changes to our system are necessary. Performance is a concern, however, since the many calls to `smalloc` in each thread will contend for the lock that guards the heap.

Alternatively, using the shadow stack approach to hold the sensitive local variables requires each thread to have its own shadow stack, just as each has its own traditional stack. The shadow stack pointer, which in single-threaded programs is simply a global variable, must therefore be stored into *thread-local storage*. Each thread has a pointer to its own shadow stack. The stack space for the thread is allocated during the first use of the shadow stack and freed when the thread terminates.

Since the same function may be called in different threads, each function with sensitive local variables retrieves the shadow stack pointer for the current thread upon entry to the function. When the pointer is updated, it must be stored into thread-local storage. For programs using POSIX threads, we add the following to the beginning of

each function body:

```
void * stackPointer = pthread_getspecific(scrash_stack_key);
stackPointer += sizeof(struct function_shadow);
pthread_setspecific(scrash_stack_key, stackPointer);
```

and before each return statement, we add:

```
stackPointer -= sizeof(struct function_shadow);
pthread_setspecific(scrash_stack_key, stackPointer);
```

Note that the structure `function_shadow` holds the contents of all sensitive local variables for that function.

The first part of the structure's name identifies the function in which it is used.

An alternative to thread-local storage would be to add an extra parameter to every function that holds the address of the current thread's shadow stack. Unfortunately, it is often not possible to change the signature of every function, since functions such as event handlers and signal handlers are called by underlying systems.

3.6.2 setjmp / longjmp

A naïve implementation of the shadow stack will not correctly handle `setjmp` and `longjmp`. These functions are frequently used as a mechanism to pass control non-locally as an interprocedural goto, which is useful for error handling. The `setjmp` call saves the register contents, including stack pointer and program counter, in a `jmp_buf` structure. A `longjmp` call takes a previously populated `jmp_buf` as an argument and restores the registers saved in this structure. Since restoring the `jmp_buf` replaces the stack pointer and program counter, the stack is unwound and the program returns to the site of the `setjmp` call, this time with a non-zero return value from `setjmp`.

Scrash maintains its shadow stack by pushing a new frame upon entry to a function and popping it just prior to exiting the function. However, the default `longjmp` implementation is unaware of the Scrash shadow stack, and will not properly restore the shadow stack pointer as it does the regular stack pointer.

We address this problem by using CIL to introduce a new structure, `scrash_jmp_buf`, which replaces a regular `jmp_buf`. It has two fields: one to contain the old `jmp_buf` structure and one to store the shadow stack pointer. We then search for all calls to `setjmp` and `longjmp` and replace them with functions that properly maintain the shadow stack pointer in addition to the registers in `jmp_buf`.

Note that when calling `setjmp` in a threaded environment, we store the thread-specific shadow stack pointer (normally stored in thread-local storage) in the `jmp_buf`. This transformation is necessary because a thread's state in Scrash is described by the contents of the registers, stack pointer, and shadow stack pointer, all of which must be stored in `jmp_buf` for `longjmp` to work properly. On a `longjmp` call, we restore the stack pointer back into thread local storage.

3.6.3 Sensitive function arguments

The C calling convention places all arguments to a function on the call stack. Thus, calling a function with a sensitive value will place sensitive information on the unprotected call stack. Our solution to this problem does not require any effort on the part of the programmer; instead, a Scrash transformation converts a sensitive argument into a pointer reference to the sensitive data. Thus, the sensitive value is never placed on the call stack. Naturally, all such function bodies, declarations, and call sites need to be modified. To transform the call site, we first allocate space on the sensitive stack for any sensitive arguments. Then, we make a copy to preserve the call-by-value semantics of C and call the function with a pointer to the data.

Rewriting a function is not possible if the program exports a fixed API, passes a function pointer to a library callback function, or has a variable number of arguments. If Scrash detects that the address of a particular function is ever passed as an argument, it will refuse to modify that function, since changing its signature could yield unpredictable behavior. Instead, Scrash prints a warning advising the user of the security vulnerability. It is then up to the developer to modify the API to avoid passing sensitive variables by value.

4 Evaluation

We tested our system by applying the Scrash code transformations to a set of open-source applications and then comparing the behavior of each modified program to that of the original. We chose our set of test applications to include commonly-used graphical and command-line programs that handle significant amounts of user data.

Our first graphical test application was `gnomecal`, the calendar portion of the GNOME Personal Information Management suite. This application consists of about 25,000 lines of C code. Our other GUI-based test application was J-Pilot, a desktop organizer application for Palm OS-based handheld computers that contains about 42,000 lines of C code. It provides support for datebook, address storage, memo and "to-do list" handheld applications, while also facilitating PC-to-handheld data synchronization and backup. Both `gnomecal` and J-Pilot use the GTK+ graphical user interface libraries. When instrumenting these programs, we first examined the source code to determine which library I/O routines were most likely to be involved in the processing of sensitive user data. We then included ap-

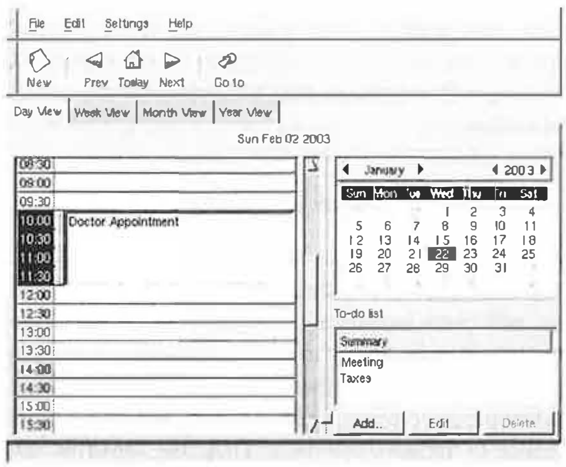


Figure 5: A screenshot of the GNOME Calendar application running with the Scrash transformations.

propriate declarations of these functions in a pre-annotated header file (as described in Section 3.2) prior to performing sensitivity inference on the program source code.

We chose the OpenSSH secure shell client, which contains about 39,000 lines of C code, as our command-line test program. For this application, it was necessary to treat all data typed by the user at the keyboard as sensitive. The password used to set up a secure connection is the most obvious sensitive value, but even after the connection is established, the client may send passwords and other private information to the server. Therefore, we again used pre-specified annotations to mark all data returned by `read` (among other functions) as sensitive.

After Scrash ran its compile-time type inference on our test applications, 24% and 10% of the stack variables used by `gnomecal` and J-Pilot, respectively, were marked as possibly containing sensitive data. For `ssh`, this figure was 59%.

We instrumented our Smalloc library to record the size and sensitivity of each run-time memory allocation request issued during the lifetime of a program. We then used each of the test applications for brief session. The run-time values from these tests are listed in Table 1. We only counted allocations performed by the application and not by any linked, precompiled libraries; this issue is discussed further in Section 5. The overall percentages of memory operations that dealt with sensitive data were lower for the graphical applications than for `ssh`. In `ssh`, the insensitive heap contains a few control structures representing the internal state of the connection, while the majority of the heap allocations are for sensitive user data that is to be transmitted over the network. We argue that the connection data is more relevant for debugging than the data being transmitted.

```
core.normal.dirty:

000732e0: 6d80 0608 7fd0 0708 0cf3 ffbf 0004 0000  m...............
000732f0: 0200 0000 34f7 ffbf e854 0908 98f8 ffbf  ....4...T.......
00073300: 6842 0908 1800 0000 a066 2440 6162 7261  hB.......f$@abra
00073310: 6361 6461 6272 6100 5842 0908 f058 0908  cadabra.XB...X..
00073320: c830 0840 c4ef 0f40 7c3b 0908 28f4 ffbf  .0.@...@|;..(...
00073330: 28f4 ffbf 5842 0908 0000 0000 8855 0908  (...XB.......U..

core.smalloc.dirty:

0006a330: 70d0 2340 0000 0000 0000 0000 0000 0000  p.#@............
0006a340: 70d0 2340 0904 0000 0100 0000 0df0 edfe  p.#@............
0006a350: 6162 7261 6361 6461 6272 6100 0000 0000  abracadabra.....
0006a360: 80d3 2340 0000 0000 70d0 2340 0000 0000  ..#@....p.#@....
0006a370: 70d0 2340 0000 0000 90d3 2340 0000 0000  p.#@......#@....

core.smalloc.clean:

0006a330: 5858 5858 5858 5858 5858 5858 5858 5858  XXXXXXXXXXXXXXXX
0006a340: 5858 5858 5858 5858 5858 5858 5858 5858  XXXXXXXXXXXXXXXX
0006a350: 5858 5858 5858 5858 5858 5858 5858 5858  XXXXXXXXXXXXXXXX
0006a360: 5858 5858 5858 5858 5858 5858 5858 5858  XXXXXXXXXXXXXXXX
0006a370: 5858 5858 5858 5858 5858 5858 5858 5858  XXXXXXXXXXXXXXXX
```

Figure 6: Excerpts from the core file of an induced crash in the `ssh` client. The top core file excerpt shows the stack with the password present – "abracadabra" from an unmodified `ssh` client. The middle core file is from a version of `ssh` that has been modified using the Scrash transformations and annotations. The password now resides in the secure region, but since the cleaning process has not yet been executed on the core file, the password is again present. The bottom core file shows that the cleaner overwrites the secure region, and all occurrences of the password have been removed.

4.1 Security evaluation

We examined core files produced by our modified version of `ssh` to verify that sensitive information was placed only in the secure region and that the cleaning process properly eliminated sensitive data. Figure 6 shows the excerpts from three core files in which we induced a program crash. The top core file is the original version of `ssh`, in which the password is present on the stack. The middle core file is the result of running `ssh` after applying the Scrash transformations, in which the password resides in the secure heap. The final excerpt shows the result after running the cleaner.

4.2 Performance

Finding privacy-relevant and performance-critical applications proved to be a rather tricky exercise for us. Many of the applications for which one would be concerned about leaks of personal data were interactive: editors, browsers, information management tools, or remote access programs like `ssh`. In the course of testing, we found that the Scrash transformations did not reduce the responsiveness of the interactive applications we tested. In an attempt to better quantify the performance impact of Scrash, we ran two tests: one real application and a micro-benchmark to illustrate worst case behavior.

To test Scrash against a privacy-sensitive program that also has performance requirements, we chose to transfer

Size (bytes)	GNOME Calendar number of requests	GNOME Calendar percent sensitive	J-Pilot number of requests	J-Pilot percent sensitive	OpenSSH client number of requests	OpenSSH client percent sensitive
0 - 1023	4216	86.9%	5914	26.7%	2073	97.6%
1024 - 2047	9	77.5%	0	–	46	100%
2048 - 3071	1	0%	1	100%	67	100%
3072 - 4095	7	85.7%	1	100%	3	100%
4096 - 5119	2	50%	2	100%	50	100%
5120 - 6143	1	100%	0	–	3	100%
6144 - 7167	0	–	0	–	3	100%
7168 - 8191	7	100%	0	–	2	100%
8192 - 9215	0	–	0	–	1	100%
9216+	9	100%	0	–	12	100%
Total	4252	86.9%	5918	26.8%	2260	97.8%

Table 1: The number and size of all run-time memory allocations (`smalloc`, `scalloc`, `srealloc`) performed by our test applications during a brief test run and the percentage of these allocations that handled sensitive data. We only count allocations done by the applications and not by any libraries that they use.

Sensitive locals moved to:	**Elapsed Time(s)**	**Increase over baseline**
Heap	27.24	33%
Shadow stack	21.71	6%
Baseline	20.51	–

Table 2: Time needed for `scp` to transfer 100 megabytes of data to a server on the same machine. The results demonstrate that using a shadow stack gives much better performance than storing sensitive locals in the heap.

Sensitive locals moved to:	**Heap allocs**	**Elapsed Time(s)**	**Increase over baseline**
Heap	657393865	242.64	373%
Shadow stack	2	62.42	22%
Baseline	1	51.28	–

Table 3: Results of running the greatest common divisor (GCD) microbenchmark. We computed 50 million GCD computations on integers. The other two implementations place the sensitive local variables on the heap and shadow stack. The first column indicates the number of heap allocations that the microbenchmark makes. The results demonstrate that using a shadow stack gives much better performance than storing sensitive locals in the heap.

large files using `scp`. This program has many of the same privacy vulnerabilities as `ssh`, as it in fact calls the `ssh` executable. The program is non-interactive, allowing us to measure changes in performance easily. For the tests, we transferred a 100-megabyte file to a server on the same machine to eliminate network-induced performance variability.

The second test is a microbenchmark that exercises the call stack heavily. We wrote this recursion-intensive benchmark to expose the worst case performance of Scrash, since every function entry and exit requires intervention from Scrash. Furthermore, all locals are declared to be in the sensitive stack, increasing the normal memory access times. The microbenchmark computes the greatest common divisor of 50 million pairs of random numbers, where each number is between 1 and 10 million. The benchmark uses Euclid's algorithm, which admits a natural recursive implementation. Each invocation performs very little computation: a modulus call, two comparisons and assignments, and then a recursive call. Due to the heavy use of recursion, the amount of stack maintenance overhead that this microbenchmark incurs is significantly greater than that of a typical application. We used the same random seed when testing all implementations, processing 657,393,863 function calls per test run. To exercise the Scrash transformations, we marked all local variables as sensitive, as well as one global variable that we used to track the number of function calls.

We performed the above tests on a 1.5 GHz Pentium 4 with 1 gigabyte of RAM, running a Linux 2.4.18 kernel with gcc 2.95. All tests were run with optimizations turned on at -O3. The tests were conducted under three different configurations: without Scrash (the baseline), using Scrash

to place all sensitive local variables on the heap, and finally using the shadow stack to hold the sensitive local variables. The results, shown in Tables 2 and 3, are based on the averages of three separate test runs per configuration. See Section 3.4.2 for a description of the two Scrash configurations.

Our initial strategy of moving sensitive stack variables to the heap via a call to `smalloc` at the beginning of each applicable function, as described in Section 3.4.2, resulted in a large performance penalty of 33% overhead for `ssh` and 373% for the microbenchmark. The microbenchmark suffers a larger overhead because it performs over 600 million allocation and free calls – one for every procedure entry. It also incurs a much higher percentage increase over the baseline because function entry time is a larger percentage of the CPU time for the microbenchmark than for `ssh`.

The second strategy, using Scrash transformations to implement a shadow stack, added much less overhead: 22% for the GCD microbenchmark and 6% overhead for `ssh` – a moderate overhead for the realistic application scenario. This overhead is a result of maintaining the shadow stack pointer at the beginning and end of the function, as well as the extra level of indirection required to access local variables.

We see that the shadow stack gives much better performance than placing sensitive locals on the heap. Consequently, we enable the shadow stack by default.

We conclude that Scrash adds only a minimal performance overhead to real applications.

5 Discussion

In addition to the runtime overhead imposed by Scrash, the system requires some effort from the programmer. This effort includes annotating an initial set of sensitive variables or deciding to use a pre-annotated "prelude" file that automatically marks the parameters and return values of certain functions as sensitive. In addition, it was necessary to make 33 lines of source code changes to `ssh` before it could run through the Scrash transformation, due to the fact that CIL is more restrictive in type checking than `gcc`. Such changes included fixing missing or mismatched variable declarations.

The performance of the Scrash code transformation tool is adequate. It takes roughly three minutes to run the entire Scrash transformation on `ssh`, from preprocessing through program modification, using the same test machine as above.

One feature of the `ssh` code was particularly problematic for Scrash: all calls to `malloc` are performed using a wrapper function, `xmalloc`, that checks for a null return value. Recall that Scrash rewrites calls to the `malloc` function to use `smalloc`, locating the new region on either the secure or insecure heap as appropriate. Since the `ssh` program calls the `xmalloc` wrapper, the only instance of `malloc` in the `ssh` source code is within the `xmalloc` wrapper. Scrash must choose whether to translate this `malloc` call into an allocation on the secure or insecure heap at compile time. Since the results of this allocation are assigned to some variables declared with the `$sensitive` keyword, Scrash conservatively translates the `malloc` call to allocate all its storage on the sensitive heap. As a result, all heap allocations in `ssh` would normally appear on the sensitive heap. To avoid this problem, we replaced the `xmalloc` function with an equivalent preprocessor macro at each allocation point. Thus, in the post-processed file, there is now one `malloc` call where each `xmalloc` call previously appeared, allowing the different `malloc` calls to be assigned to different heaps.

We must be a bit careful in evaluating the success of a technique like Scrash. For example, the absence of the password from the core file does not mean that there is no sensitive information related to the password in the core file. It may be possible to ascertain the size of a sensitive buffer by comparing pointers. If p is a pointer to a sensitive data field, an attacker can bound the size of the sensitive data by comparing all other heap-allocated pointers, t, to the sensitive data pointer:

$$\min_{t>p}(t-p)$$

That is, the size of the buffer at p is at most the difference between p and the first pointer whose value is greater than p. Thus, it may be possible to reveal the length of a variable-sized sensitive buffer even if all variables that explicitly store this length are kept in the sensitive memory region. This apparent vulnerability would seem to suggest that p is also sensitive and should be placed on the sensitive heap, adding an extra level of indirection to all accesses to p. We eschew this extra indirection, however, in favor of providing greater debugging usefulness to the developer, since hiding the pointer values may hamper the developer's ability to track down memory problems.

Another issue with Scrash involves the use of precompiled and dynamic (shared) libraries. Current libraries such as `glibc` are written without consideration of the concept of sensitive data. CQual understands the semantics of many `glibc` functions and will correctly propagate qualifiers across, for example, calls to `memcpy`. There is no way, however, for a source-level translation like Scrash to modify the storage of variables in precompiled libraries. For example, a precompiled version of `strcpy` may keep a char temporarily on the stack, or `strlen` may keep a running string length count as a stack variable. In the event of a crash, these variables will remain on the insecure stack, where they can leak pieces of sensitive information. One solution would be to recompile libraries with Scrash under the assumption that all data passed to a shared library is sensitive. The library would therefore use the shadow stack and

sensitive heap so that sensitive data may be passed to the shared library without fear of privacy violations. However, we have not implemented this solution in our prototype.

As we discussed in Section 2.1, there are tradeoffs between user privacy and utility to the developer when dealing with crash information. Scrash provides the developer with a larger set of tradeoffs than the all-or-nothing choice that exists currently, while requiring minimal effort and time to specify and apply these tradeoffs to a program.

We believe that Scrash will help developers to allay users' privacy concerns about using crash reporting tools, and dissuade users from turning off the automatic crash reporting features in their applications. Widespread use of remote crash reporting will aid developers in improving the overall quality of software, in addition to helping make users aware of software patches for problems that they are experiencing.

6 Related Work

To the best of our knowledge, there has been no previous research published on the topic of limiting crash data to ensure privacy. The only other sources to mention this issue are the aforementioned Department of Energy advisory about Microsoft's Dr. Watson [12] and an online article on the same subject [18]. Both sources suggest that the user should disable crash reporting altogether to avoid a privacy risk.

Dr. Watson [4], the independent BugToaster application for Windows [2], the Bug-Buddy bug reporting tool for GNOME [1] and the Talkback quality reporting agent for Netscape/Mozilla [3] represent the current state of the art in remote crash reporting software. All are capable of sending back portions of the program's memory contents, including the registers, call stack and heap. Bug-Buddy is the least automated of the four, starting automatically when a GNOME program fails but then requiring a high degree of user participation to send a crash report. The other three require only the consent of the user via a dialog box to send a crash report.

The core file cleaning process is analogous to the scrubbing process that Gutmann advocates for securely deleting sensitive information from media, such as RAM or magnetic media [10]. His cleaning process is aimed at protecting against physical attacks against storage media that are not easily erasable. Other work focuses on creating a large block of erasable memory from a much smaller block using cryptographic techniques to achieve similar ends [6]. In contrast, we view our cleaner as operating on the *contents* of files to eliminate sensitive information so that they may be safely sent over the network.

There is a large body of work that describes techniques for efficient allocators [17] and garbage collectors [16]. Region-based memory allocators in which multiple heaps are exposed have also been studied [7, 8]. While they present a richer set of semantics than we need, these sources helped to inspire our implementation. We used the Vmalloc software release as the basis for Smalloc, our secure memory allocator [15]. Vmalloc provides an alternative allocator to `malloc` that exposes many different allocation fit strategies and provides rich internal interfaces.

We use CQual, a static analysis tool, to track the possible spread of sensitive information [14]. Sabelfeld and Myers [13] survey language-based systems for statically tracking information flow in a secure manner. Tracking information flow typically involves removing covert channels within a program, which can require extensive code modifications. While information-flow concerns are a central theme of this work, we do not address the issue of convert channels.

7 Future Work

Changes to Scrash in the short term mostly involve improvements to the analysis phase. The implementation of CQual that our current system uses is at times too conservative – it marks too many variables as `$sensitive` – but we expect to be able to use a more accurate version soon. The new implementation, currently under development, will use a polymorphic analysis of functions so that more variables can be safely labeled insensitive. Modifying Scrash to work with C++ is another area of active interest; CQual has recently been extended to work with C++ code.

In addition, we hope that support for Scrash will be incorporated into some of the standard bug reporting tools, such as the GNOME Bug-Buddy. Another avenue would be to combine runtime error detection tools, such as StackGuard or CCured [5, 9], with Scrash. When these runtime tools would detect a violation, Scrash would send a core file to the developer. This pairing would aid in the detection of security vulnerabilities such as buffer overruns.

8 Acknowledgments

Many people have contributed to this project. Dan Wilkerson and Rob Johnson implemented many last-minute CQual features for us, while John Kodumal, Jeff Foster, and the rest of the CQual team provided advice on using CQual. We thank Ben Liblit, David Gay, and Jeremy Condit for their insightful comments and suggestions. Finally, David Wagner provided helpful guidance along the way.

References

[1] Jacob Berkman. Project Info for Bug-Buddy. `http://www.advogato.org/proj/bug-buddy/`,

2002.

[2] Bugtoaster. Do Something about Computer Crashes. http://www.bugtoaster.com, 2002.

[3] Netscape Communications Corp. Netscape Quality Feedback System. http://wp.netscape.com/communicator/navigator/v4.5/qfs1.html, 2002.

[4] Microsoft Corporation. Dr. Watson Overview. http://www.microsoft.com/TechNet/prodtechnol/winxppro/proddocs/drwatson%_overview.asp, 2002.

[5] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, January 1998.

[6] Giovanni Di Crescenzo, Niels Ferguson, Russell Impagliazzo, and Markus Jakobsson. How to Forget a Secret. In *Proceedings of Symposium on Theoretical Aspects of Computer Science*, number 1563 in Lecture Notes In Computer Science, 1999.

[7] David Gay and Alexander Aiken. Memory Management with Explicit Regions. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 313–323, 1998.

[8] David Gay and Alexander Aiken. Language Support for Regions. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 70–80, 2001.

[9] Scott McPeak George C. Necula and Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In *Principles of Programming Languages*, 2002.

[10] Peter Gutmann. Secure Deletion of Data from Magnetic and Solid-State Memory. In *Sixth USENIX Security Symposium Proceedings*, 1996.

[11] George C. Necula, Scott McPeak, Westley Weimer, Raymond To, and Aman Bhargava. CIL: Infrastructure for C Program Analysis and Transformation. http://www.cs.berkeley.edu/~necula/cil, 2002.

[12] U.S. Department of Energy Computer Incident Advisory Capability. Office XP Error Reporting May Send Sensitive Documents to Microsoft. http://www.ciac.org/ciac/bulletins/m-005.shtml, October 2001.

[13] Andrei Sabelfeld and Andrew C. Myers. Language-Based Information Flow Security. *IEEE Journal on Selected Areas in Communications*, January 2003.

[14] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *10th USENIX Security Symposium*, pages 201–220, August 2001.

[15] Kiem-Phong Vo. Vmalloc: A General and Efficient Memory Allocator. *Software Practice & Experience*, 26:1–18, 1996.

[16] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In *Proc. Int. Workshop on Memory Management*, number 637, Saint-Malo (France), 1992. Springer-Verlag.

[17] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic Storage Allocation: A Survey and Critical Review. In *Proc. Int. Workshop on Memory Management*, Kinross Scotland (UK), 1995.

[18] Brandon Wirtz. Dr. Watson's a Big-Mouth. http://www.griffin-digital.com/dr__watson.htm, 2002.

Implementing and testing a virus throttle

Jamie Twycross
jamie@milieu3.net

Matthew M. Williamson
matthew.williamson@hp.com

Hewlett-Packard Labs, Bristol, U.K.

May 13, 2003

Abstract

In this paper we build on previous theoretical work and describe the implementation and testing of a *virus throttle* - a program, based on a new approach, that is able to substantially reduce the spread of and hence damage caused by mobile code such as worms and viruses. Our approach is different from current, signature-based anti-virus paradigms in that it identifies potential viruses based on their network behaviour and, instead of preventing such programs from entering a system, seeks to prevent them from leaving. The results presented here show that such an approach is effective in stopping the spread of a real worm, W32/Nimda-D, in under a second, as well as several different configurations of a test worm.

1 Introduction

CNET dubbed 2001 the "*Year of the Worm*" [14] and 2002 has only seemed to confirm this, with high-speed mobile code forming the majority of entries in the top ten of most infectious programs. Indeed, on the weekend this paper was being completed, the W32/SQLSlam-A worm [22] infected what is currently estimated at 75,000 machines in 30 minutes [16] and caused major network disruption. Finding effective ways to prevent such activity is a high priority.

The work we present here provides just that - an extremely effective way of halting the propagation of certain classes of worms. Our approach, *virus throttling* [27], [30], is based on the observation that under normal activity a machine will make fairly few outgoing connections to new machines, but instead is more likely to regularly connect to the same set of machines. This is in contrast to the fundamental behaviour of a rapidly spreading worm, which will attempt many outgoing connections to new machines. The idea behind the virus throttle is to put a rate limit on connections to new machines such that normal traffic remains unaffected but suspect traffic is slowed, quickly detected and stopped.

Our approach is different from current approaches in *three* key ways: it focuses on the *network behaviour* of the virus and prevents certain types of behaviour, in our case the attempted creation of a large number of outgoing connections per second. It is also unique in that, instead of stopping mobile code from entering a system, it restricts the code from *leaving*. Lastly, because connections over the allowed rate are *delayed* and not dropped, the system is tolerant to false positives and is therefore robust.

In this paper we give a detailed description of an implementation of the virus throttle and our experimental setup, and present the results of a number of tests. These tests show that the virus throttle is able to very quickly detect and prevent worms spreading from an infected machine. For example, the throttle is able to stop the W32/Nimda-D worm [21] in under *one* second. Furthermore, since the throttle prevents subsequent infection, the effect on the global spread of the virus will depend on how widely it is deployed. Our results also show that when 75% of machines are installed with the throttle, the global

spread of both real and constructed worms is substantially reduced. Throttled machines do not contribute any network traffic in spite of being infected, significantly reducing the amount of network traffic produced by a virus.

The next section, Section 2, paints the background against which our work stands, briefly reviewing what mobile code is and how it propagates and discussing current approaches to limiting this propagation. It then goes on to outline the foundations on which our perspective is based, and briefly reviews related work. Section 3 describes in detail the design and implementation of our virus throttle, the performance of which, along with our experimental setup, we describe in Section 4. Concluding remarks are made in Section 5.

2 Background

In this section we offer a brief review of mobile code and current approaches to limiting its spread. We then go on to introduce the conceptual framework upon which our work rests, and end the section with a summary of related work.

2.1 Mobile code

We are interested in a class of software broadly known as *mobile code* [10]. For our purposes we define mobile code pragmatically as any program that is able to transfer itself from system to system with little or no human intervention. Many examples of such mobile code can be found in real life, the most common of which are the many viruses and worms that are becoming an increasingly prevalent feature of the Internet [5, 14]. While mobile code can propagate through different media, for example, removable storage, since we are particularly interested in propagation across *networks*, we will restrict our discussion to code that spreads across this medium. Although technical difference do exist between virus and worms [6, 7], in what follows we will use these terms and mobile code interchangeable.

An archetypal piece of mobile code can in general be seen as repeating a cycle composed of several distinct stages. The code will perform some form of *scan* to attempt to locate target machines which are vulnerable to infection, and will then attempt to *exploit* any target machines found. If successful, the exploit will allow the mobile code to *transfer* a copy of itself to the target machine, which will itself begin its own scan/exploit/transfer cycle.

2.2 Current approaches

Current approaches to virus protection involve preventing a virus from entering a system, predominantly through signature-based detection methods [8]. These methods concentrate on the physical characteristics of the virus i.e. its program code, and use parts of this code to create a unique signature of the virus. Programs entering the system are compared against this signature and discarded if they match. In terms of the three-stage scan/exploit/transfer cycle described above, current approaches can be seen as focusing on the *transfer* stage.

While this approach has up to now been fairly effective in protecting systems it has several limitations which, as the number of virus samples increases, decrease its effectiveness. It is fundamentally a reactive and case-by-case approach in that a new signature needs to be developed for each new virus or variant as it appears. Signature development is usually performed by skilled humans who are only able to produce a certain number of signatures in a given time. As the number of viruses increases, the time between initial detection and release of a signature increases, allowing a virus to spread further in the interim. Furthermore, comtemporary viruses are using techniques such as polymorphism and memory-residency to sidestep signature detection entirely.

2.3 Agents and complex systems

An alternative and fruitful approach can be gained by viewing the mobile code as an autonomous agent acting within a complex system [19]. Such a paradigm shift leads to an emphasis on different concepts and also allows a vast amount of literature on complex and adaptive systems from fields concerned with these entities to be drawn upon. For example, when viewed as an agent the question of how the agent behaves within the environment it inhabits becomes as important as the purely mech-

anistic details of its construction on which the current approach described in the last section is based.

The distinction between mechanism and behaviour can lead to some simple but potentially powerful conclusions. While a virus is able to instantiate an effective spreading mechanism in an extremely large number of ways, each requiring a separate signature, the number of ways in which a virus can behave to spread effectively is perhaps much more limited. This is especially the case with the class of high-speed worms which are becoming increasingly prevalent and which, due to their high-speed nature, need to scan a large number of hosts per second. Behaviour is a much more powerful discriminator than that employed in current, mechanistically-orientated signature-based methods, as it potentially allows the automatic identification and hence removal of an entire class of worms.

Considering behaviour also leads to some less obvious insights, one being that it could be more productive to focus on preventing viruses leaving a system, as opposed to stopping them entering, the strategy taken by current methods. While such a seemingly altruistic approach may at first sight appear ineffective, recent work by Williamson and Leveille [29] and other work discussed in the next section indicates that it can be extremely effectual in preventing the spread of viruses across networks.

2.4 Related work

Our approach is related to the "*behaviour blocking*" of Messmer [15] which seeks to specify policies defining normal or acceptable behaviour for applications. If an application breaches such a policy it is reported to an administrator. The approach we take differs in that it is able to automatically respond to abnormal behaviour, taking the administrator out of the loop in this respect. The benign facet of this response is particularly important as it makes the throttle more tolerant to false positives. Another example of a benign response used in an intrusion detection application is given by Somayaji and Forrest [20], although their application implements this response in relation to abnormal sequences of syscalls. Bruschi and Rosti [2] discuss various ways in which hosts can be prevented from participating in network attacks and describe a tool, AngeL [3], which can be used to prevent systems from participating in such attacks. AngeL, however, relies on a signature-based algorithm to detect attacks, inspecting network packets for predefined sequences of data, for example, shellcode or unusual HTTP requests, in contrast to our behaviour-based approach.

3 The virus throttle

After outlining the context of our work in the last section, we now go on to give a detailed explanation of the design and implementation of a virus throttle, of which an initial description and proof-of-concept based on theory and simulation was presented in [27]. The main focus of this section will be on a TCP implemention of the virus throttle, although we have also tested UDP, SMTP and Exchange throttles with similar designs.

3.1 Design

The virus throttle is a program that limits the rate of outgoing connections to new machines that a host is able to make in a given time interval. For the purposes of simplicity in this section we will assume that the host has *one* unique address - its source IP address, although the implementation described below allows for multiple source IP addresses. Connections to a remote machine are established through what is known as a three-way-handshake in which the initiator of the connection, the source machine, sends a TCP SYN packet to the target machine, identified by a destination IP address. The target machine then sends back a SYN-ACK packet, which the source machine replies to with an ACK packet [17]. By controlling the number of SYN packets transmitted from the source machine we can control the number of connections it is able to make.

A note should be made about the relationship between connection attempts and SYN packets. At the application layer, a connection is usually initiated by opening a socket [23]. This results in the sending of an initial TCP SYN packet and, if no response is received within a certain time, the sending of further, identical, SYN packets. This continues up to a maximum time, the socket timeout, when the socket will give up and return control to the application. Thus, in attempting to open a single connection a machine may actually transmit several SYN packets. In our implementation we make

no attempt to differentiate initial SYN packets from retries, and count the retries as separate connection attempts.

A machine will establish many such connections in the course of normal usage, for example when requesting a web page or the delivery of email. Many worms also use such connections when scanning in order to establish the existence and configuration of remote machines, with high-speed worms such as W32/Nimda-D [21] or CodeRed [4] initiating large numbers of connections to different targets per second. The virus throttle rests on the observation that the patterns of connections due to normal usage are very different from the patterns of connections created by such mobile code. Our research has suggested that under normal usage often no more than *one* connection to a target not recently connected to is made per second, and that the majority of connections are made to destination addresses that have recently been connected to [27].

The virus throttle parses all outgoing packets from a machine for TCP SYN packets. The destination address of an intercepted SYN packet is then compared against a list of destination addresses of machines to which connections have previously been made, which we term the *working set*. The working set can hold up to 5 such addresses. If the destination address is in this working set the connection is allowed immediately. If the addresses is not in the working set and the working set is not full i.e. it holds less than 5 addresses, the destination address is added to the working set and the connection is once again allowed to proceed immediately. If none of these two conditions are met, the SYN packet is added to what we term the *delay queue* and is not transmitted immediately.

Once every second the delay queue is processed and the SYN packet at its head and any other SYN packets with the same destination address are popped and sent, allowing the establishment of the requested connection. The destination address of this packet is also added to the working set, the oldest member of which is discarded if the working set is full. If the delay queue is empty at processing time and the working set is full, the oldest member of working set is also discarded, allowing for the potential establishment of one connection per second to a target not recently connected to.

This design, summarised schematically in Figure 1, allows hosts to create as many connections per second as they want to the 5 most recently connected-to machines. Any further connection attempts will be delayed for at least a second, and then attempted. Delaying connections rather than simply dropping them is important as such a benign response [20] allows the virus throttle a certain amount of leeway in its conception of normal behaviour and a response that, if incorrectly targeted at legitimate connection attempts, will introduce an often imperceptible delay in the connection, instead of prohibiting it entirely.

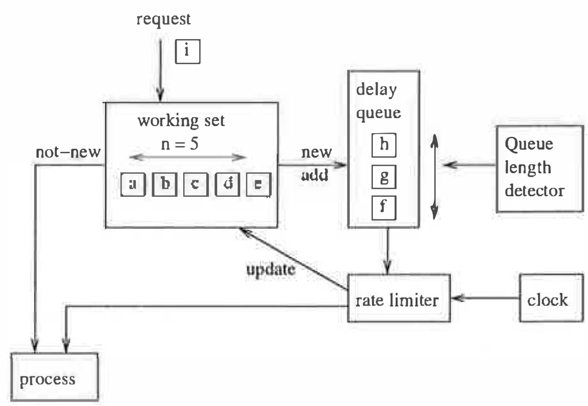


Figure 1: A schema of the control flow of the virus throttle

In [27], it was argued that if the rate of connections to new targets is high, as is the case with many worms, the delay queue would rapidly grow resulting in attempted connections being severely delayed. When implementing the throttle, we took this idea further and introduced an upper limit to the size of the delay queue which, once reached, will disallow all further connection requests by the host. Thus the throttle can be said to behave benignly within certain limits. From observation of the normal behaviour of a range of users we saw that the delay queue rarely grew bigger than a handful of packets and concluded that the size of the delay queue could offer an indication of the presence of a worm on the host: large delay queue sizes would almost certainly indicate an application behaving in a suspect manner. In our throttle we set this upper limit on the delay queue size to 100 packets.

Lastly, as mentioned previously, we have also implemented UDP, SMTP and Exchange versions of the throttle. The UDP throttle works in exactly the same way as its TCP counterpart except that instead of parsing outgoing network traffic for TCP SYN packets, it looks for outgoing UDP packets,

each of which is considered a separate connection attempt. The email throttle is described fully in [28].

3.2 Implementation

In this section we describe how the virus throttle was implemented, beginning with a short overview of the structure of the Linux network stack. Necessarily, this section is fairly technical, and the reader is referred to more complete works such as [1, 18] for further information.

The Linux network stack strictly consists of two data structures, the ptype_all linked list and the ptype_base hash table, containing pointers to packet handler functions [13], but, at a conceptual level, can simply be thought of as a list of packet handler functions. Outgoing packets are placed onto the list by a packet handler and traverse the list until they reach its head, after which the packets are passed to the hard_start_xmit function supplied by the appropriate network device driver for transmission on the wire. The virus throttle works by replacing the pointer to the hard_start_xmit function registered by the network device driver with a pointer to a function of its own. This means that, since the hard_start_xmit function is called every time a packet has finished traversing the network stack and is ready to be transmitted on the network interface, we are able to intercept and control every packet leaving the network device. In essence, the throttle can be viewed at this level as an ethernet device driver wrapper.

To accomplish this wrapping the virus throttle is implemented as a Linux 2.4.18 kernel module which, in its init_module routine, fetches the pointer to the net_device structure registered by the current network device driver. The net_device structure contains entries to pointers to the packet handling functions registered by the network device driver, including the hard_start_xmit function, whose pointer we replace with a pointer to our own transmit function.

Our own transmit function is a simple packet parser which looks for TCP SYN packets, the packets used to establish stream-based connections between applications. If the packet being parsed is *not* a TCP SYN packet, it is passed on to the original hard_start_xmit function for transmission as usual. If, however, it is a TCP SYN packet it is, as described in the previous section, either allowed to pass immediately, in which case it is handed off to the original hard_start_xmit function and possibly added to the working set or, if the working set is full, is added to the delay queue for later transmission. Both the working set and delay queue data structures are instantiated as linked lists using the linked list structure provided by the Linux kernel. The working set list stores the destination address of the packet, while the delay queue list stores the source and destination addresses of the packet, a copy of the sk_buff data structure associated with the packet, and the time it was enqueued. The limit on the size of the delay queue is implemented by monitoring the size of the delay queue in the packet parser and setting a flag if this size increases beyond the specified upper limit. The packet parser will never allow a connection attempt to commence if this flag is set.

Processing of the delay queue is handled by a kernel thread which wakes up a specified number of times per second, in our case once per second. The delay queue is then processed as described in the previous section, with packets deemed suitable for transmission dequeued and passed on to the original hard_start_xmit function. Due to the fact that the delay queue and working set are data structures shared by the enqueueing packet parsing routines and the dequeueing delay queue routines, these two data structures are carefully protected by spinlocks.

In order to allow for the fact that one host may have several different IP addresses or that the virus throttle may be placed on some intermediate system such as a bridge or gateway and hence see packets transmitted with a number of different source IP addresses, our implementation actually keeps an array of working set and delay queue data structures. Each entry in this array corresponds to a working set and delay queue for a particular source IP address.

4 Testing

Now that we have detailed the design and implementation of the throttle, in this section we move on to describe how we evaluated its performance. Initially, we outline our experimental setup and then go on to present the experiments we performed and their results.

4.1 Experimental setup

In order to effectively test our throttle in a range of different scenarios we first had to develop a secure testbed on which the virus throttle could be exposed to real and constructed mobile code. It is this testbed that we now describe, more details of which can be found in [25].

4.1.1 Testbed

The testbed consists of a rack mounted single-chassis HP Blade Server bh7800 [12] providing the physical infrastructure for 3 separate LANs and housing what includes three 25-port 10/100 switches and 16 bh1100 Server Blades. Each Blade has a 700MHz Pentium III processor with a 30GB hard-disk, onboard graphics and 3 network interfaces. One network interface is provided to the management LAN by a remote management card (RMC) daughterboard, while the other two provide connections to the remaining two LANs.

As well as being physically separate, the functional roles of the three LANs have also been separated. The *management LAN* provides access to the RMC daughterboard on each Server Blade which can be used for tasks such as power-cycling the Blade. The second LAN is designated the *administrative LAN* and is used to handle the installation and configuration of the Blades, and the coordination of experiments and collation of data from these experiments. The third LAN, the *experimental LAN*, is the network on which the experiments are actually performed. This setup is summarised in Figure 2.

In addition to the Blade Server, we are also employing four 1.8GHz Pentium 4 boxes, two of which act as servers on the administrative and experimental LANs respectively. The administrative LAN server, as well as running services such as PXE, DHCP and FTP necessary for the configuration of the Server Blades also coordinates our experiments. A third machine acts as our data collector and is in essence a network sniffer. A monitoring port [11] has been configured on the switch on the experimental LAN and the sniffer machine, running tcpdump [24], listens on this port and keeps a copy of all traffic on this LAN. The fourth machine is our infector which, at the start of some of our experiments, we bring up and use to inject a copy of the virus under study into

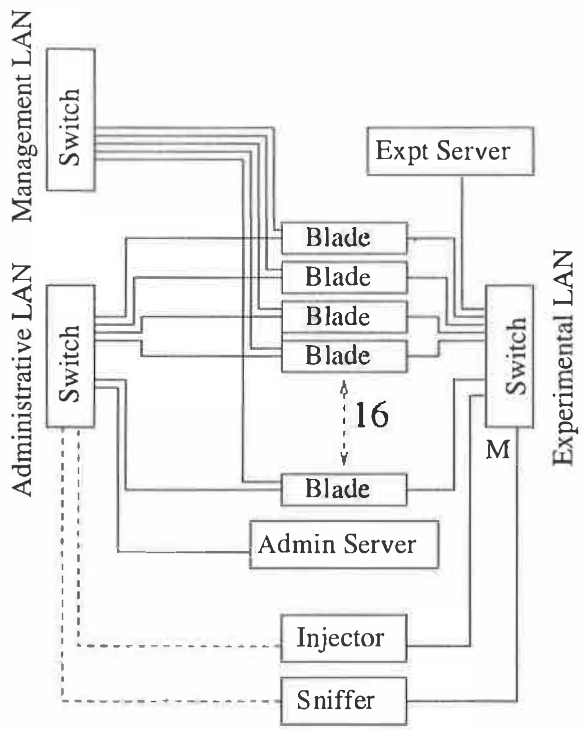


Figure 2: The configuration of the testbed

one of the machines on the experimental LAN. This machine is then brought down and plays no further role in the experiments. For coordination purposes, the sniffer and infector machines have network connections to both the experimental and administrative LANs, and particular care has been taken to ensure that no traffic from one of these LANs is able to cross to the other LAN.

A slimmed-down Redhat 7.3 Linux installation with a custom 2.4.18 kernel forms the operating system on the Server Blades. As the experiments involve a variety of operating systems and settings, to ease configuration and save time we run VMware Workstation 3.2 [26] on each Server Blade. VMware allows the running of one operating system, the guest operating system, inside another operating system, the host operating system, in our case Linux. For the experiments below we use a standard installation of Windows 2000 Professional as the guest OS with VMware configured in bridged-only networking mode. Setting a VMware guest OS to use bridged-only networking means that the guest will largely use its own network stack, as opposed to that of the host OS if host-only networking is configured. This results in more realism in the network behaviour of applications running under the guest

OS but, since bridged-only networking employs a proprietary VMware driver located fairly low down in the Linux network stack, places constraints on the implementation of the virus throttle not necessarily present in a production version of the throttle.

This configuration of the testbed allows us to automate the process of running experiments to a large extent, essential when it is necessary to repeat experiments a number of times to reduce the noise inherent in the random nature of the programs we are studying. Obviously, when a network has machines known to be infected with viruses, the security and isolation of this network is of prime importance. As well as the measures described above and careful configuration and testing of all software, the testbed is housed in a secure laboratory physically preventing connections to all other networks and strict policies concerning data transfer are imposed.

4.1.2 Test worm

In order to test the effectiveness of the virus throttle at allaying the spread of worms which scan at different rates a test worm was developed. The test worm consists of a basic stream-socket server [23] which listens for connections on a specified port. When it receives a connection attempt the server starts a scanner whose properties, such as scan rate and address range, can also be specified. This scanner then scans the IP address space by attempting to make connections to addresses on the port the test worm server is listening on. As the type of scanning used is TCP connect scanning [9], if the scanner discovers the IP address of a machine running a test worm server it will trigger the server to start the scanner on this machine. This scheme allows a fairly realistic simulation of the scan/exploit/transfer worm lifecycle but, since no actual exploit or file transfer is involved and since any machines infected by the worm have, *a priori*, to already have been infected with the worm, guarantees that the test worm will never spread autonomously.

4.2 Results

Using the testbed setup described above we have been able to securely observe a variety of different viruses spreading across the experimental network and have performed a number of different experiments with the virus throttle, which we now go on to detail.

4.2.1 Stopping speed

We were first interested in the time it would take worms scanning at a number of different rates to cause the delay queue to reach its upper limit, 100 packets in our implementation. Remembering that once the delay queue has reached this limit we disable all further connection attempts, this time is effectively the time it takes the throttle to stop a worm. The experimental LAN on the testbed was configured with two machines, a gateway and a Server Blade running a Windows 2000 Professional guest OS. Table 1 records the time taken for the delay queue to reach 100 and the number of connection attempts made before this time when the Blade Server is infected with the real W32/Nimda-D virus [21] and the test worm configured to scan at various rates. As this table shows, the virus throttle takes only 0.25 seconds to stop Nimda, which scans at a rate of around 200 connections per second, from spreading, and under 5 seconds to stop any application that scans at a rate of 20 connections or more per second. In the Nimda case, the throttle only allows one packet out on to the wire before networking is shut down, and a maximum of 5 packets for the test worm configured to scan at 20 connections per second. The stopping times for the SQLSlammer worm [22] of a UDP implementation of the throttle have also been included in Table 1 and show that the throttle is able to stop this worm in 0.02 seconds.

4.2.2 Mobile code propagation

In order to test the effectiveness of the virus throttle at reducing the propagation of a real worm, the testbed was set up with one class C network containing 16 Server Blades running a VMware-encapsulated copy of Windows 2000 Professional vulnerable to the W32/Nimda-D virus [21]. A machine which acted as the default gateway and DHCP server for this network was also configured. One of the Windows machines was then infected with W32/Nimda-D and its progress through the network observed by gathering data using the sniffer configured as described above. This data was then analysed and the time at which each system became

connections per second	stopping time	allowed connections
Nimda		
120	0.25s	1
Test Worm		
20	5.44s	5
40	2.34s	2
60	1.37s	1
80	1.04s	1
100	0.91s	1
150	0.21s	0
200	0.02s	0
SQLSlammer		
850	0.02s	0

Figure 3: Average time taken by the throttle to stop real and test worms

infected with the Nimda virus determined. By varying the number of machines on the network which had the throttle installed and by repeating each experiment 10 times, we were able calculated the average time taken for a given number of machines to be infected assuming that a certain percentage of the machines on the network had virus throttles installed. These results are shown in Figure 4.

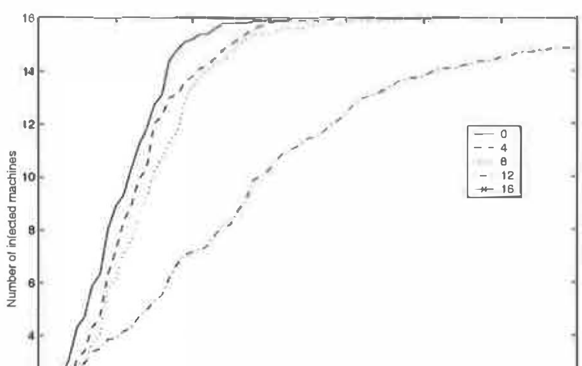


Figure 4: Infection times for different numbers of installed virus throttles (Nimda)

As Figure 4 shows, when no throttles are installed on any of the 16 machines Nimda takes on average just over 20 minutes to spread to all 16 machines. This time is slightly reduced when 25% and 50% of machines have throttles installed. However, when at least 75% of machines are installed with the throttle, Nimda is only able to spread to on average around half the machines on the network in the same amount of time, and has not spread to all 16 machines even after over 70 minutes. This represents both a decrease of 50% in the number of machines infected and a substantial increase the time taken for the worm to spread. When every machine is installed with the throttle, the worm is unable to spread at all.

Aside from the damage caused by the malicious payloads of many high-speed worms, these worms often cause denial-of-service attacks through the amount of network traffic they generate. The Sapphire worm caused network uplinks to become saturated due to the sheer quantity of traffic infected machines generated, whereas Nimda, while generating substantially less quantities of network traffic, caused routers and firewalls to fall over due to their inability to process the increased number of connection requests generated by infected machines. Figure 5 shows, for the experiment described at the start of this section, the traffic load over time on the test network. Here, the effect of having a network protected by the throttle is even more marked, with an approximately 25% reduction in viral traffic when only 25% of the network in protected by the throttle. When 50% of the machines are throttled the traffic load is reduced by over half, and when 75% are throttled to around 10% of its unthrottled average. Having all machines installed with the throttle quickly reduces all viral traffic to *zero*. These results show the ability of the virus throttle to substantially reduce the amount of network traffic generated by mobile code.

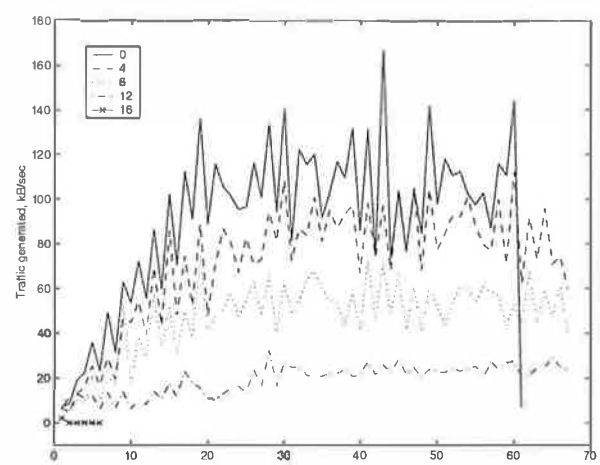


Figure 5: Traffic loads for different numbers of installed virus throttles (Nimda)

4.2.3 User trials

While the previous section shows that the virus throttle is extremely effective in slowing and stopping the spread of certain classes of worms, an equally important aspect in determining its overall utility is establishing its impact on the performance of applications which legitimately establish connections in the course of normal user behaviour. If the throttle, for example, prevents users from accessing their emails, or slows this down to an unacceptable speed, it will soon be uninstalled.

From preliminary tests in which the throttle was installed on a number of researchers' machines, we received no reports of impaired network performance. We are currently involved in a series of much larger user trials in which the throttle is installed on the gateway used by a considerable number of users running a wide range of network-capable applications under several different operating systems. Initial results also point to no noticeable degradation in network performance.

To further explore the interaction of the throttle with legitimate network-aware applications, we have also implemented a throttle simulator which takes packet traces as input. To gather these packet traces we configured a monitoring port on a 80-port switch on HP's internal network and ran a packet sniffer on this port. This has allowed us to gather large quantities of data over extended periods of time from a range of different machines. Separating out these traces on a machine-by-machine basis and then running them through the simulator allows us to rapidly assess the applicability of the throttle to different classes of machines running different services. Preliminary results from the simulator show that the majority of network traffic from a throttled machine passes onto the network undelayed, and that none of the large quantities of legitimate network traffic we have gathered is mistaken for viral traffic. A fuller discussion of the applicability of our approach is given in [27].

5 Conclusion

In this paper we have, after presenting the necessary background, described in detail the implementation and testing of a virus throttle. From the tests described above we have been able to show that the virus throttle is highly effective in detecting, slowing and stopping both real worms such as W32/Nimda-D and a test worm configured to scan at different rates. Our results also show that the virus throttle can substantially reduce the global spread of a worm, and hence the amount of network traffic produced.

In conclusion, this paper has demonstrated virus throttling to be a powerful tool in the prevention of high-speed worm propagation. We believe that throttling, when combined with current signature-based methods, will be an essential ingredient in any multilayered anti-virus solution.

References

[1] D. P. Bovet and M. Cesati. *Understanding the Linux kernel.* O'Reilly & Associates, 2002.

[2] D. Bruschi and E. Rosti. Disarming offense to enable defense. In *Proc. of the New Security Paradigms Workshop 2000*, pages 69–75, Ireland, 2000.

[3] D. Bruschi and E. Rosti. AngeL: A tool to disarm computer systems. In *Proc. of the 2001 Workshop on New Security Paradigms*, 2001.

[4] CERT Advisory CA-2001-19. CERT Coordination Center. http://www.cert.org/advisories/CA-2001-19.html.

[5] D. Chess. The future of viruses on the Internet. Presented at the Virus Bulletin International Conference, October 1-3, 1997.

[6] F. Cohen. Computer viruses - theory and experiments. In *Proc. of the 7th Security Conference*, pages 143–158, 1984.

[7] F. Cohen. A formal definition of computer worms and some related results. *Computers and Security*, 11(7):641–652, 1992.

[8] F. Cohen. *A short course on computer viruses.* John Wiley & Sons, Inc., 1994.

[9] fyodor. The art of port scanning. *Phrack Magazine*, Volume 7, Issue 51, 11 of 17, 1997.

[10] R. A. Grimes. *Malicious mobile code.* O'Reilly & Associates, 2001.

[11] HP Procurve Series 2500 switches - management and configuration guide. Hewlett-Packard Company, 2000.

[12] HP Blade Server bh7800 service guide. Hewlett-Packard Company, 2002.

[13] kossak and lifeline. Building into the Linux network layer. *Phrack Magazine*, Volume 9, Issue 55, 12 of 16, 1999.

[14] R. Lemos. Year of the worm. http://new.com.com/2102-1001-254061.html.

[15] E. Messmer. Behaviour blocking repels new viruses. http://www.nwfusion.com/news/2002/0128antivirus.html.

[16] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. The spread of the Sapphire/Slammer worm. http://www.cs.berkeley.edu/ nweaver/sapphire/.

[17] J. Postel. Transmission control protocol. RFC 793, DARPA, 1981.

[18] A. Rubini and J. Corbet. *Linux device drivers.* O'Reilly & Associates, 2001.

[19] H. G. Schuster. *Complex Adaptive Systems.* Scator Verlag, 2001.

[20] A. Somayaji and S. Forrest. Automated response using system-call delays. In *Proc. of the 9th USENIX Security Symposium*, pages 185–197, 2000.

[21] W32/Nimda-D. Sophos Anti-Virus. http://www.sophos.org/virusinfo/analysis/w32nimdad.html.

[22] W32/SQLSlam-A. Sophos Anti-Virus. http://www.sophos.org/virusinfo/analysis/w32sqlslama.html.

[23] W. R. Stevens. *UNIX network programming.* Prentice Hall, 1990.

[24] tcpdump. http://www.tcpdump.org.

[25] J. Twycross. Studying mobile code: an experimental setup. Technical report, Hewlett-Packard Labs, 2002.

[26] VMware Workstation 3.2. VMware Inc. http://www.vmware.com.

[27] M. M. Williamson. Throttling viruses: Restricting propagation to defeat malicious mobile code. In *Proc. of the ACSAC Security Conference*, Las Vegas, Nevada, 2002.

[28] M. M. Williamson. The design, implementation and testing of an email throttle. Submitted to the Annual Computer Security Applications Conference, Las Vegas, N.V., 2003.

[29] M. M. Williamson and J. Leveille. An epidemiological model of virus spread. To appear in the Proceedings of the Annual Virus Bulletin Conference, 2003.

[30] M. M. Williamson, J. Twycross, J. Griffin, and A. Norman. Virus throttling. In *Virus Bulletin*, U.K., 2003.

Establishing the Genuinity of Remote Computer Systems

Rick Kennell & Leah H. Jamieson
Purdue University School of Electrical and Computer Engineering
{kennell,lhj}@purdue.edu

Abstract

A fundamental problem in distributed computing environments involves determining whether a remote computer system can be trusted to autonomously access secure resources via a network. In this paper, we describe a means by which a remote computer system can be challenged to demonstrate that it is *genuine* and trustworthy. Upon passing a test, it can be granted access to distributed resources and can serve as a general-purpose host for distributed computation so long as it remains in contact with some certifying authority. The test we describe is applicable to consumer-grade computer systems with a conventional network interface and requires no additional hardware. The results of the test can be conveyed over an unsecured network; no trusted human intermediary is needed to relay the results. We examine potential attacks and weaknesses of the system and show how they can be avoided. Finally, we describe an implementation of a genuinity test for a representative set of computer systems.

1 Introduction

For most types of valuable real-world objects, there are generally accepted methods of assessing their genuinity in a non-destructive fashion. Archimedes determined that King Heiron's crown was not made of pure gold by noticing that the mass and volume of the crown did not match the known density of gold. We can discern real diamonds from imitations by examining their electrical characteristics and their indices of refraction. We can determine that money is not counterfeit by carefully studying its watermarks and other identifying features.

Unfortunately, we have few such measures for computer systems. When answering the question of whether a computer system is real, we can only verify that it looks like a computer and acts like a computer. Unfortunately the dynamic nature of a programmable computer means that it may not always behave the same in the future. Furthermore, when that computer system is moved from our immediate presence, we have few guarantees that it has not been physically modified or reprogrammed.

We introduce the need for a remote system genuinity test with a motivating example: Suppose Alice is the conscientious administrator of a network of computer systems that rely on a central NFS [42] server. Only trusted systems are allowed to act as NFS clients. Bob and Mallory use the systems in the network for large distributed applications that manipulate data on the file server. In addition to accomplishing normal work, Mallory would like to either steal or corrupt Bob's data by subverting an NFS client. Mallory has a deadline to perform an especially large computation and has made an arrangement with a distant colleague to borrow several hundred new computers to join the computer network temporarily in order to assist with the work. This necessarily means that they must also be able to access his data on the NFS server.

How can Alice determine which systems should be given access to the NFS server? She could travel to the location of each of the new systems in order to configure and physically secure them, but this might take more time than she is able to commit. Allowing Mallory to specify the systems eligible to be NFS clients would give him an opportunity to deliberately misconfigure a system in order to allow him to access Bob's data. Knowing the identity of the remote systems gives Alice no assurance of how the systems will behave.

What Alice really wants is an automated way of determining that a remote system has been properly configured before it is granted access to the NFS server. For the specific case of NFS, "properly configured" means that only Alice has administrative control over the client. In a broader sense, it means that the system's hardware and software will act in a deterministic fashion that corresponds with Alice's expectations. To measure this determinism, it is necessary for Alice to be able to decide whether the system's hardware and software are *genuine*. For instance, it is easy to see that the correct system software running in the context of a simulator can not always be trusted because it cannot be known

whether the simulator will always act the same as real hardware. Moreover, anyone who controls the simulator will be able to spy on or manipulate the data resident in the simulated environment. The same lack of trustworthiness is more obviously apparent for compromised software running on real hardware.

We demonstrate a method by which we can simultaneously measure the genuinity of a system's hardware and software, remotely, without the need for a trusted human intermediary. This method can be used to decide on a level of trust that can be imparted to the system in question. Our mention of NFS (without special encryption or credential support) in the previous example is deliberate. In Section 5 we describe our implementation of a genuinity test that allows us to trust a remote system for use as an NFS client. Although attacks against our method are possible, their risk may be acceptable in some environments.

This paper is organized as follows. In Section 2 we examine the basic methods for determining the genuinity of software, hardware and combinations thereof. In Section 3 we show how the results of such a test can be delivered through an insecure network to an observer. Section 4 describes several forms of attack against the system as well as our methods for reducing the risk of those attacks. Section 5 shows various results of an implementation of our system. We conclude our paper with an overview of related work in Section 6 and a summary of our results and future plans in Section 7.

2 Tests of Genuinity

In order for some secure authority to be able to trust that a remote computer system will always act in a deterministic manner, it is necessary to verify two things:

- The computer must be a real computer—not a simulator or emulator. If the computer was not genuine, as in the case of a simulator, the environment would then be subject to potential manipulation by the simulator's controller, whereas a genuine system would be governed only by the definition of the machine's instructions.

- The computer must be running exactly the software that the secure authority expects. If the software environment is known completely then the behavior of the system can be known as well, but without perfect knowledge of the running software, the action of the system is potentially unpredictable.

We first examine the challenge of building a test for the correct software while assuming that the hardware can be trusted. We then integrate that test with a microprocessor genuinity test. This combined test constitutes an architecturally-sensitive execution fingerprint.

2.1 Software Genuinity

In order to introduce the approach to verifying the genuinity of a system's software, we consider a naïve example of some computational *Entity* running a program that must check that its own instructions have not been tampered with. (We assume throughout this paper that the program to be verified is an OS kernel.) The intent might be to demonstrate to an *Authority* elsewhere on a network that the program is correct. If we can make a closed-world assumption that the software to be tested exists entirely in a known section of memory, we can accomplish this by including a subroutine in the program that would compute a checksum of that memory space. The subroutine can send the outcome to the Authority so that it can be compared with a known-good result. Any attempted forgeries are easily detected by using a cryptographically-secure hash to compute the checksum so that the likelihood of changing the instructions to produce a correct checksum is extremely small.

The possibility of a replay attack is reduced if the Authority is allowed to challenge the Entity by specifying one or more subregions of the program's instruction address space. Such a method has been used for several collaborative network applications to verify the integrity of the client software. For instance, the AOL Instant Messenger service can discriminate against rogue clients by requesting a cryptographic hash of a section of the client program's text segment [3, 38].

It is a more difficult problem to guarantee that the code that was executed to compute the checksum was really the subroutine in the program. For instance, consider the possibility that the Entity contained the expected program in its memory but executed a rogue program from some other location. Detecting an attack of this nature is equivalent to determining whether a piece of code is presently executing at a specific location. A test of this nature is an example of *Software Tamperproofing* [12, 21]. Normally, this technique is useful for situations where, for example, one wishes to ensure that a piece of software cannot be illicitly duplicated in another application. Here, we want to use it to ensure that the code we are interested in has not been modified to run in any place other than the one in which it was intended [13]. There are several architecturally-sensitive ways of constructing such tests using modern microprocessors, and

we will address this thoroughly in Section 2.4.

A basic limitation of our closed-world assumption is that we cannot incorporate dynamically-allocated data or any other non-static information into the checksum in a meaningful fashion. However, if we can determine that the instructions of the software are correct (and that these instructions are being executed) we may be able to trust the software to further interrogate and verify the integrity of its data. We will address this in further detail in the context of our implementation in Section 5.1.

2.2 A Microprocessor Genuinity Test

In order to determine the genuinity of a system's hardware, we first make a limiting assumption that the primary item of importance is the microprocessor specifically rather than trying to look at the entire system. In order to determine a microprocessor's genuinity, one must exercise a representative subset of its functions to check that it fully conforms to all of its specifications. In conducting such a test, it is very easy to discriminate between microprocessors that have different instruction set architectures, but it is considerably harder to tell the difference between two implementations of the same architecture. Nevertheless, we maintain that there are always differences between implementations that can be observed by software at some level even if the instruction set is the same. For instance, in determining the difference between two microprocessor implementations that differ only by their cache geometry, an instruction sequence with a particular memory reference pattern would take longer to execute on one of the processors. If the microprocessors had performance counters capable of measuring the number of cache misses or if they had direct means of evaluating the cache contents, a test could be constructed to discriminate between the two implementations without resorting to timing analysis.

Determining the difference between a real microprocessor and a simulator is an apparent contradiction to Turing's thesis [45] that any computer is theoretically capable of any mechanical calculation—including simulation of another computer. However, even modern high-performance simulators are typically one or more orders of magnitude slower than real computers [32]. If the Authority establishes a time limit for the Entity's response, it will be able to ascertain whether the Entity was being run in the context of a simulator. Furthermore, by constructing the checksum using operations that are most difficult to simulate, the disparity in execution time between simulator and real computer is maximized. The goal then is to establish a repertoire of tests that execute much more slowly on a simulator than on the real computer.

In general, any test to be used for the purpose of determining genuinity will exercise a function of the CPU that has the following ideal characteristics:

- The function will occur automatically as a side-effect of instruction execution. This ensures that a simulator will be forced to do multiple jobs at once—execute instructions as well as accurately model their lower-level architectural impact.
- The function must be deterministic and predictable. The Authority must still be able to either obtain the result of a genuinity test from a trusted reference platform or (slowly) simulate and predict the result on its own.
- The effects of the function can be easily measured. This allows the genuinity test to run as efficiently as possible on the real CPU.
- The function will have a good deal of implicit parallelism. This minimizes the chance that a simulator will be able to mimic it quickly.

Consider the previous example of a CPU that includes hardware performance counters that can measure the occurrence of cache misses. For such a CPU, modifications to the cache's state will occur as a side-effect of all load and store operations. In the case of a miss, there is some implicitly parallel activity that occurs to determine the cache line that should be replaced, and this parallelism is largely hidden from the instruction set architecture. If we understand the cache geometry and replacement policy we can accurately predict, for a particular access pattern, which cache lines will be replaced and when, but doing so generally requires an effort proportional to the *associativity* of the cache in question. At the conclusion of a correctly-executed test that exercises this pattern, the hardware counter should contain the number of misses we predicted to occur, and the final state of the cache should also be as we predicted.

The memory hierarchy makes an excellent device to satisfy the characteristics of a good test since it has traditionally been found difficult to simulate precisely [7, 33], and, when memory is modeled at its fullest detail, the result is that the simulator runs at least an order of magnitude slower [32, 49] than if it had a memory model adequate only for simulation of the instruction set architecture. We expect this disparity in speed to only increase in the future as CPU implementations continue to

use memory hierarchies with higher levels of associativity and more complicated replacement policies.

2.3 A Combined Software/Hardware Genuinity Test

By incorporating instructions that characterize the system's execution into the checksum procedure, we can build a mechanism that can check the integrity of its own instructions while simultaneously ensuring that the instructions are running on a real computer. Each load that the checksum procedure performs has several side effects: cache lines are evicted and refilled, translation lookaside buffer (TLB) entries are evicted and refilled, and bus transactions are performed. For each group of instructions fetched, another TLB is consulted for virtual memory mapping of the instruction space. Each branch in the checksum procedure is predicted by the CPU to be either taken or not taken. Many sources of meta-information about the CPU's execution exist; however, our selection criteria indicate that the best candidate function is the memory hierarchy. In particular, the TLB is a good choice since, for most architectures, it has a higher associativity than the caches. It also has a deterministic replacement policy whose effects can be easily measured on most platforms.

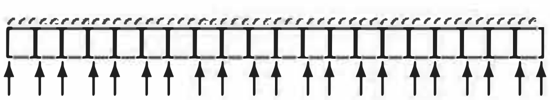


Figure 1: Linear memory traversal

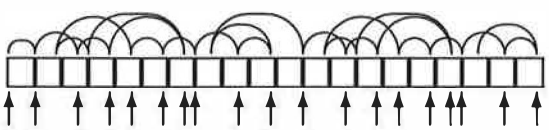


Figure 2: Pseudorandom memory traversal

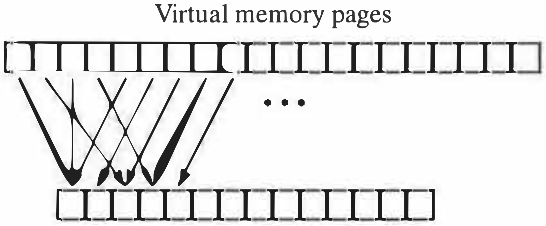


Figure 3: Pseudorandom memory mapping

In addition to making the genuinity test difficult to simulate quickly, we also need to make sure that the architectural sensitivity is inseparable from the memory checksum and that the results from one genuinity test give no clues to an observer as to how a future test will be computed. A checksum procedure that traverses memory in a linear fashion would be unacceptable since it would cause a data TLB (DTLB) replacement pattern that could be predicted without simulating the DTLB. Figure 1 graphically depicts a linear access pattern across pages of memory. Pages of memory are shown as blocks and the access pattern is shown by the small arcs above. The TLB misses predictably occur on the first access to each page. We need to introduce as much run time complexity as possible into the test; therefore, we construct the checksum so that it traverses memory in a pseudorandom fashion as in Figure 2. Doing so creates a greater run time uncertainty about whether the page of the word that is loaded is presently mapped by the DTLB. We further complicate the test by aliasing the pages of the physical memory region we intend to check multiple times through a much larger virtual region (Figure 3) rather than using a one-to-one mapping from virtual space to physical space as is customary. Doing so increases the duration of the checksum and constitutes a better discriminant against the possibility of a simulation attack. By varying the pseudorandom walk and the mapping from virtual to physical space for each new invocation of the genuinity test, the checksum result will be different each time and unrelated to other invocations.

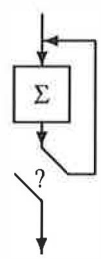


Figure 4: Simple checksum representation

As the checksum procedure makes the pseudorandom walk through the virtual region, it consults a hardware performance counter to determine if the previous load caused a hit or miss in the DTLB. This information is incorporated into the checksum result, and the procedure continues in this manner until the pseudorandom walk is complete.

It is important that one or more sources of meta-information about the procedure's execution be incorporated continuously into the checksum result in order to ensure that the checksum of the memory contents and the test for genuinity of the execution environment cannot be separated. This combination has a benefit of al-

lowing us to relax the requirement of a cryptographically secure hash for computation of the memory checksum. Each word read from memory by the checksum procedure can be added to the running result as long as the DTLB hit/miss information is incorporated in a non-additive manner. For instance, after each checksum addition the DTLB miss count could be XOR-ed into the result. Since the memory checksum and the DTLB miss count are not linearly related, the integrity of the memory checksum is enforced. Graphically, we can depict this simple procedure as in Figure 4 where the checksum component might consist of

```
sum = sum + [ PseudoRandomLocation ]
sum = sum XOR TLB_MissCount
```

and the decision element simply determines whether the last value in the pseudorandom sequence has been read.

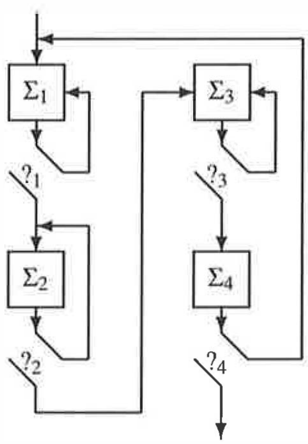


Figure 5: Complex checksum representation

Reducing the frequency of integration of the TLB miss count would not reduce the burden of an attacking simulator to continually track the TLB contents. The procedure could be easily modified to incorporate the TLB miss count into the checksum after every few iterations. This would allow the genuinity test to run more efficiently with only a minor risk of separability of architectural sensitivity from the checksum calculation, but it would still require almost the same computational overhead for an attacking simulator. Furthermore, if several unrelated sources of meta-information were available, each source could be incorporated into the checksum in a round-robin fashion at various points in the genuinity test. Adding more information sources would only require an attacking simulator to continuously simulate more functions that already occur automatically in the real CPU. For instance, if the procedure were partitioned into multiple code sections spread out over several pages of memory as in Figure 5, the instruction TLB (ITLB) would register a hit or miss each time a new code section was invoked.

Further refinements to the checksum procedure shown in Figure 5 can be made, such as making the decision points dependent on the checksum (rather than only the pseudorandom location pointer) or making each of the checksum and decision points contain unique functionality.

The end result is a memory checksum of the running application which has been perturbed by meta-information about the execution of the procedure. The perturbation is difficult to simulate in a timely manner due to the typically large associativity of the TLBs. If either the memory contents or the execution of the procedure is not correct, the incorrect result will be computed. If the procedure is evaluated by a simulator, it cannot be done quickly enough to meet the Authority's deadline. The result will differ for each combination of pseudorandom walk and virtual-to-physical memory mapping thereby allowing an Authority to challenge each new Entity with a unique test every time. Since the checksum procedure cannot discern at run time whether the result is correct, its result constitutes a *zero-knowledge proof* that the system is a real computer executing the expected software. The result can then be delivered to an Authority that can compare it to a precomputed, known-good result to determine the genuinity and, therefore, the trustworthiness of the system.

2.4 Additional Requirements of the Genuinity Test

There are several additional aspects of the genuinity test that are important for the veracity of the test. In particular, some steps must be taken to ensure that the execution environment is sufficiently isolated from outside influence. Since the mechanism presented relies on the fact that only the memory references pertaining to the checksum are to be used to predict the behavior of the DTLB, there can be no other memory references performed by the procedure. Therefore, all temporary data needed by the procedure must be held in CPU registers for the duration of the test. Furthermore, since any temporary diversion from the procedure might cause extraneous memory references, the checksum procedure must be run with all external interrupts disabled. After the genuinity test has been satisfied, interrupts are re-enabled, and the machine is allowed to function as a normal execution host.

If the secure Authority and the Entity are located far

apart, there will be some (potentially unbounded) network transport delay to issue the challenge to the Entity and return the results to the Authority. In theory, an allowable delay should be no longer than the marginal time difference between the checksum procedure running on the target CPU compared to the same procedure running on the highest performing simulator. The longer the test that we construct runs, the better this margin becomes, so a long procedure is desirable. In practice, however, we can assume that the network transport delay is small and any pathological delays will simply be treated as failures. Hopefully, such delays will be only temporary, and a new genuinity test can be retried at a later time.

The problem of determining whether the program being verified is actually running is addressed by incorporating a suitable degree of *introspection* into the verification procedure. The simplest way of doing so would be to introduce a section of the verification procedure that would incorporate the present value of the program counter (PC) into the checksum value; however, the location of the checksum code is known in advance, and it does not require much program analysis to replace each PC incorporation by a constant-valued expression in a rogue program running elsewhere. Because software running in a location other than its intended location or running as a part of another application will produce a different ITLB miss pattern than the expected one, we can employ architecturally-sensitive techniques to identify such an attack. In particular, on architectures that support such operations, we can modify the genuinity test to incrementally incorporate the contents of the ITLB into the checksum. This not only ensures that the code was run in the expected location but also improves the certainty of the test by incorporating one more piece of architectural meta-information into the checksum.

We can also rely on more complicated architectural state in order to make such an attack possible only in the context of a full microprocessor simulator. For instance:

- Some microprocessors write information to the virtual memory page tables to indicate accessed or dirty pages. For instance, an x86 CPU marks a page table entry (PTE) as "accessed" when it is read or executed. This progression of PTE modification for a particular test is deterministic and predictable and we can incorporate it into the checksum calculation by including the page tables in the memory mapping. An attacker would certainly be forced to simulate the entire virtual memory subsystem in order to keep track of this information.
- Some microprocessors allow privileged applications to directly examine the TLB's Content-Addressable Memory cell (CAM) contents. By incorporating this information into the checksum value, the outcome will only be correct if the ITLB contains values corresponding to the correct execution location.
- Some microprocessors have hardware-based last-called stacks that can be interrogated to obtain the address of the last control-modifying instruction (such as a JUMP or CALL instruction). Since our checksum algorithm has pseudorandom control-flow that is based on the contents of memory, this information is not expressible as a constant. By incorporating this information into the checksum value, we force an attacker to either simulate this hardware stack or fully simulate the instruction stream.

If the checksum obtains the correct result, it will have demonstrated that its own exit path is trustworthy. This exit path will be responsible for invoking the code that delivers the result to the Authority via the network and for confirming that critical data values of the computer system are intact (e.g., the call stack and interrupt vectors) in order to prevent any code entry points that are not already known to the OS kernel. The kernel that invokes the checksum procedure is expected to remain in operation until the end of the session in which it participates with the Authority.

3 Demonstrating Genuinity Via an Insecure Network

In our work, we assume that the distance between the Entity and Authority does not permit the establishment of a secure communication link prior to the genuinity test. Furthermore, we intend our system to be used without the benefit of a trusted human who could serve as the courier of a shared secret. In order to prevent an interposition attack by a malicious router, we can leverage the genuinity of the Entity, as indicated by its valid checksum computation, to negotiate a public key exchange with the Authority. To do so, we first ensure that the public key of the Authority is embedded into the verified memory space of the Entity's genuinity test; a properly computed checksum value will be an indication that this is true. When the Entity sends the computed checksum to the Authority it can first encrypt the result, along with an identifier chosen at random by the Entity, using the public key of the Authority. Although it is critical that the Entity deliver this information to the Authority

in a timely manner, a single public key encryption does not greatly delay the result. Thereafter, the Entity would have an identity known only to itself and the Authority. At some later point, the Entity will generate a new public key and send it to the Authority, along with the random identifier as an indication that the new key really belongs to the authenticated system and not an intermediary. The knowledge of genuinity of the remote system represents a convenient improvement to the problem of key exchange in the presence of an interlocutor [8, 15, 16, 39].

The primary difficulty in such an exchange is obtaining enough entropy to generate a random identifier that cannot be easily guessed by an adversary. Although the genuinity test must be deterministic, there are still events present in the Entity that are random. In particular, we have found that periodic sampling of the timestamp counter during the course of the test will produce sufficient entropy (even for repeated invocations of the same test code). The minor variations in memory and CPU pipeline timing force the CPU to sample the free-running timestamp counter at unpredictable intervals. Such a random number generator is similar in some respects to the TrueRand function in CryptoLib [29]. Experimental results for our random value generator are shown in Section 5.4.

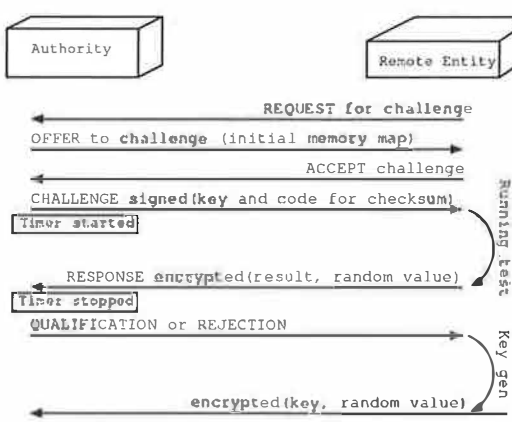


Figure 6: Network protocol

Figure 6 illustrates the following negotiation protocol initiated by the Remote Entity.

1. The kernel running on the Entity sends a network packet to the Authority containing (cleartext) information about its microprocessor and requesting a genuinity test.

2. The Authority generates a suitable test and sends a cleartext message offering to challenge the Entity with a genuinity test. The message contains the memory mapping to apply for the test.

3. The Entity sends a cleartext message to the Authority accepting the offer and initializes the virtual memory mapping.

4. The Authority sends a cleartext packet that contains the code to be run and a public key for the response. It also records the time at which it sends the packet.

5. The Entity receives the code and key, and loads both into memory. It then transfers control to the test code. When the checksum completes, it jumps directly to a (now verified) function in the Entity's kernel that encrypts the checksum result and the random quantity and sends them to the Authority. This step is carefully designed to minimize the time that the unencrypted values are resident in memory. Most likely the unencrypted values will never be visible on the external memory bus.

6. The Authority receives the reply, registers its time, and decrypts it. If the reply contains the correct checksum result and was received within an allowable time, it marks the Entity as a genuine host. It sends a reply packet indicating the Entity's qualification.

7. The Entity generates a new session key, concatenates it with the previous random value, encrypts them with the testcase public key and sends them to the Authority. Here the random value acts as an authenticator that the Authority can use to know that the session key it is receiving is from the same system that showed itself to be genuine. The session key can then be used to establish a secure virtual private network or some other kind of secure communication channel.

Note that while the kernel on the Entity is waiting for a challenge message it has no means of reliably determining the sender of a packet. Since the challenge consists of runnable code that is to be inserted into the Entity's kernel memory, it is important to first check the message's validity in order to avoid running code sent by an attacker. The risk of such an attack can be reduced by generating a second key pair and embedding the public key into the kernel that is loaded onto the Entity. The Authority can sign the messages that it sends, and the Entity can then discriminate against bogus messages. Since the genuinity test negotiation protocol is already sensitive to the exact type of kernel running, this protocol addition poses no further complications to the test other than the time needed to validate the challenge signature.

4 Potential Attacks

It is necessary to elaborate on some potential attacks against our genuinity test and ways of guarding against them in our implementation. In particular, there are several forms of simulation that could be attempted as well as some methods of ensuring that they cannot be effective. Since a primary discriminant in our method is time of execution, we are faced with the fact that there is a definite lower bound on the type of target CPU that can be verified. If a system that lies below that bound requests a genuinity test, it must be rejected. Furthermore, because the performance of new systems (on which a simulator can be run) is constantly improving, the lower bound on the target CPU is also increasing. Finally, as the state of the art in simulator construction improves, target CPUs that were once a safe bet for a genuinity test are placed into question. In order to assess an appropriate lower bound, we first need to examine the kinds of simulators that are possible.

4.1 Algorithmic Simulators

A full simulation of the instruction execution and any architectural side-effects would be too slow to pass the test that we have created. There remains the possibility that the simulator does not pretend to be a microprocessor but is custom-tailored to simulate only the checksum procedure's *effect* on the microprocessor. For instance, if it is known that a remote system with particular TLB characteristics is to be shown genuine, the effect of the test on the target CPU can be efficiently simulated by a specialized application running on a very fast machine. We refer to this as an *ideal simulator*.

Here, again, it is necessary to assert that the checksum code must be executed in order to obtain the correct result. To do so, it is necessary for the Authority to deliver the challenge to the Entity in the form of unique, executable instructions in order to prevent the simple interpretation by a dedicated algorithmic simulator. For instance, instead of delivering values indicating the configuration of the pseudorandom sequences in the challenge, the Authority must send the executable code to perform the checksum. If the checksum procedure was sufficiently intertwined with its containing application, a malicious Entity would need to either do a great deal of program analysis to quickly *construct* the simulation algorithm or use full instruction simulator to initiate the test. Such safeguards are another example in the study of software tamperproofing [12, 21].

4.2 Virtualizing Simulators

Some modern system simulators [25, 30] are capable of efficiently running an entire operating system within an application by means of a technique called virtualization. This involves running the simulated OS kernel at an artificially low privilege level in such a way that execution is trapped and simulated only when a privileged operation is attempted. This allows the majority of instructions to execute natively on the CPU. However, the checksum procedure running on a virtualizing simulator would fail after the first privilege trap since it would be necessary to save some of the CPU state and doing so would involve referencing memory. Referencing any memory would alter the TLB miss patterns and invalidate the checksum.

For many microprocessor architectures, it is also difficult to build a virtual machine monitor that does not leak information about its supervisor [40]. Assuming that such information can be obtained in an efficient manner, it too can be incorporated into the checksum procedure in order to discriminate against such a simulator.

An improvement to the idea of virtualization involves dynamic translation of the privileged instructions into subroutine calls that mimic their operation in the context of the simulator. For purposes of an attack on a checksum procedure, this reduces the complexity of the problem to that of an ideal simulator. However, our results in Section 5 indicate that even if the virtual simulator could quickly converge on dynamically-generated native code that efficiently simulated the multiple implicitly-parallel architectural features, it could not execute it quickly enough to succeed unless there was a very wide disparity between the speeds of the simulator's host CPU and target CPU.

4.3 Hardware Attacks

The ultimate attack on our genuinity test involves modifying the remote system in such a manner that it can successfully compute the correct result for the checksum procedure but still allow a third party to physically inspect and alter the system after the test has completed. This can be accomplished by attaching hardware to the coherent memory bus of a system or attaching an analyzer to the microprocessor's probe ports. For many operating environments, the risk of such an attack is acceptable since undertaking it would require significant skill and equipment.

Alternatively, one can imagine the same type of attack occurring in a symmetric multiprocessor system where

the primary CPU is unable to detect the presence of a malicious secondary CPU. Since conventional CPUs can do nothing to ensure or evaluate the security of their own external interfaces, there can be no guarantee of security in the presence of such an attack. This is an open problem in our work, and we continue to examine such a possibility.

In order for any direct attack against the microprocessor to be successful, it must either be done transparently while the machine remains in operation or done very quickly while the machine is temporarily off-line. An attacker might pause the CPU (for example: a notebook computer's suspend-to-disk operation) long enough to interrogate the memory and determine the shared secrets for the secure network connections and then substitute them into another machine that he or she controlled. To guard against this possibility, we require that the remote machine remain in active contact with the Authority with a period no longer than the time necessary to pause the machine to obtain information. If the contact time exceeds an appropriate timeout, the remote machine is no longer considered trustworthy and must go through another genuinity test before trust is restored. Similarly, the trusted software running on the remote machine should destroy its secrets if it falls out of contact with the Authority.

There is also the possibility that the user has preconfigured the Entity to allow a *hotkey* sequence to cause a subversion of the system software. Ideally, there would be no systems for which we could not disable a suspension request or some other type of subversion mechanism. Because most systems implement user input using interrupt-driven methods, we believe we can avoid most problems by carefully checking the interrupt vectors and the relevant I/O configuration for device drivers to make sure that they are valid and safe. Section 5.1 describes this process in greater detail.

5 Implementation

We created a trial implementation of our hardware/software genuinity test using the Linux 2.5 kernel as a host application. The kernel provided us with a convenient and portable environment for manipulating virtual memory and network interfaces. We initially made only minor modifications to the kernel in order to incorporate our test mechanism. In particular, we inserted a few pages of empty space near the beginning of the text segment to hold alternate page tables and added a set of functions to perform the network test negotiation. The test negotiation was set up to be invoked automatically during kernel initialization—before a filesystem existed or 'init' was started. We implemented an in-kernel 128-bit RSA algorithm to perform the public-key encryption of the return result.

Our Authority consisted of the following components:

- A generator to build RSA-128 key-pairs. Each new test used a unique public key.
- A testcase generator that randomly combined predefined code chunks into a unified image. A cache of tests were precomputed for different models of x86 microprocessors.
- A network server that handled the negotiation of genuinity challenges and selection of tests based on microprocessor type. This server also maintained an indication of the status of a remote host (e.g. whether it had been recognized as genuine yet or had been rejected).

Most specifications for our system were flexible and chosen to best illustrate the validity of our approach. For instance, a 128-bit RSA key would be too small in practice. We chose this size in order to minimize the time needed by the Entity to perform the encryption. Our RSA implementation was not optimized for speed and a larger key size would be easily substituted assuming some modest improvements to the code. We also made the assumption that the Entity to be tested would operate without non-volatile storage. The kernel was initially loaded via the network using GRUB [19] or Etherboot [36]. The key exchange at the conclusion of the genuinity test was used to negotiate keys for an IPsec [28] session. The Authority then allowed NFS exports to the Entity. It may be desirable to allow the use of a non-volatile storage device for swap space or local executables. Various methods exist for ensuring that secrets swapped to disk are not compromised by an attacker [37] and that executables on a non-volatile file system can be trusted [46]. By combining these elements, we can create a trusted system that uses data served securely by the Authority.

5.1 Securing the Non-Static Information

One disadvantage of using a Unix-like kernel as the host application for a genuinity test is that it has a great deal of non-static and allocated data that could be useful to an attacker as a means of subverting the kernel after a successful completion of the genuinity test. (Recall that only the static or predictable contents of memory

are incorporated into the genuinity test.) For instance, consider the possibility that an attacker loaded the kernel, modified a pointer in an unchecked data region and allowed the kernel to initiate the genuinity test. Later, the attacker could invoke some behavior to exploit that pointer.

The areas of the kernel in question are the non-constant data segment, the initially-zero segment (or 'BSS' in the Unix vernacular), and the driver initialization text. In addition, the kernel dynamically allocates virtual memory for its data structures. We use a layered approach to ensure the integrity of this data.

First, much of the non-constant data segment is designated non-constant only as a convenience. For instance, the syscall table is specified in an assembly language .data section even though it is (hopefully) not modified during regular kernel operation. By adding a section identifier, it was moved to the read-only data section and was subjected to the genuinity test.

When the genuinity test completes, it jumps back into a verified section of code in the kernel that will restore interrupt handling, encrypt the results and send them to the Authority via the network interface. Before interrupts are reenabled, this code is guaranteed to function deterministically and can be entrusted to perform checks on the data that could not be subjected to the genuinity test. In order to maximize the time available to perform these checks, the code encrypts the results and sends them over the network and verifies data *before* re-enabling interrupts. Items verified by this code include the interrupt vectors and root virtual memory page table. In doing so, we can verify that no malicious agents are present in memory that were placed there before the invocation of the genuinity test.

The Linux kernel moves much of the device driver initialization code into a segment of virtual memory that is freed after use. In order to verify the integrity of this code, we disabled the function that freed the space and modified the linker script to move the text of the initialization code into the kernel text section. Doing this also allowed us to later use a known-genuine system to find the result of a checksum computation for another system.

We made an attempt to invoke the genuinity test as early as possible in the kernel's initialization stage to avoid the proliferation of allocated data. Nevertheless, because relaying of the test results requires a working Ethernet adaptor as well as system bus and chipset functionality, some data structures must be allocated to support access to them. One way of eliminating this data would be to restart the kernel initialization process at a point before any memory was made allocatable.

5.2 Precomputation of Checksum Results

In order to be able to quickly certify a remote Entity, our Authority must precompute checksum results for each genuinity test. This can be done by using an off-line simulator that is able to do an adequate job of emulating the architectural features of the target microprocessor. This may take a significant amount of time, but can be amortized over the expected running time of the certified remote hosts. The Authority need only maintain enough results to handle an initial surge of requests for genuinity tests.

In some cases, genuinity tests may exploit instruction execution side-effects that are deterministic, but are unpredictable. For instance, if the microprocessor has an undocumented TLB replacement policy it may not be possible to construct a simulator to use for result computation. In such cases, the Authority can use an existing genuine system to compute the results for new tests. When doing so, it is important that the exit path of the checksum algorithm does not cause a re-initialization of a known-genuine execution host, but we also do not want to open the possibility for a non-genuine host to intercept its checksum results and use them for an interposition attack.

We solve this dilemma by adding a flag to the challenge delivered by the Authority that indicates whether the Entity should deliver the results over the network and reinitialize itself in preparation for becoming a known-genuine host. The exit path of the checksum algorithm encrypts the checksum result and random identifier and then checks this flag. If the flag is not set the system destroys the value of the checksum result and random identifier but preserves the encrypted version and then returns to its prior execution environment. The encrypted version of the information is only useful to the Authority, and the random identity serves as a *nonce* to prevent reuse of the values. The Authority can use a secure channel to both initiate a new test and procure the result. The only side effect on the remote host will involve a momentary pause of other activity while its interrupts are disabled.

By allowing the existing base of genuine hosts to compute the results of new tests, the Authority can generate tests for new systems on a demand basis. It is also much easier to use one CPU to compute results for another equivalent CPU than it is to construct a suitable simula-

tor. In practice, we precomputed most of our checksum results in this manner.

5.3 Benchmarks

In order to establish a time limit for the response delay for a correct checksum result we used simulators to evaluate a particular genuinity test. It is relatively easy to tell the difference between a fast CPU and a simulator running on a similarly-fast host CPU. Therefore, we chose a particularly conservative example where we used a 133MHz Intel Pentium as a target CPU. We ran our simulators on a 2.4GHz Intel Pentium 4 system. In order to sustain an artificial performance advantage of the real CPU over the simulators, we incorporated as much architectural meta-information as possible. We used the Authority software to generate the following random genuinity test:

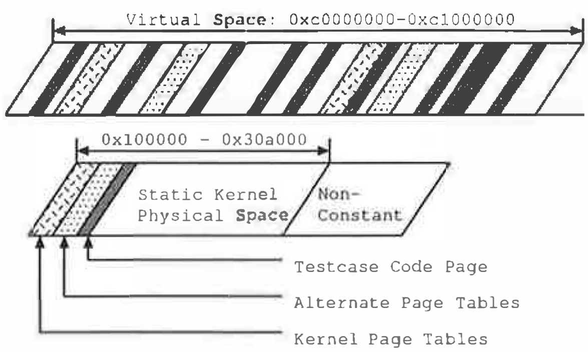


Figure 7: Memory Map for Benchmark Testcase

- The static region of the kernel's physical memory (roughly 0x00100000 – 0x0030a000) was mapped into the 16MB region from 0xc0000000 – 0xc1000000 using the alternate page tables in the beginning of the kernel text segment. This was a pseudorandom mapping that favored the specific page where the checksum code was loaded. On average, most pages of the kernel were aliased by three virtual pages. The code page was aliased by 2661 virtual pages. The kernel page tables, as well as the alternate page tables were mapped in the virtual memory space. Figure 7 depicts the mapping. Note that this 16MB region is the only memory accessible for the duration of the testcase. Instructions, page tables and data are all visible in this space. The self-contained nature of this region makes it very difficult to subvert the genuinity test by patching in malicious code since there is no accessible place that it can be hidden.
- The checksum code was constructed in nodes spread over 22 of the code page mappings in virtual memory. Each of the nodes branched to one of two other nodes on the 22 pages. The sections of code were offset so that they did not overlap in the single physical code page that they were all aliased to.
- The checksum code nodes were selected as follows:
  - One node accessed the ITLB CAM cells to determine whether an instruction fetch for a selected target would be a hit or miss. In the case of a hit, the physical page and replacement information were extracted from the CAM cell. This information was incorporated into the checksum value.
  - One node made a similar access to the DTLB CAM cells.
  - One node accessed the tag and replacement information for the data cache and instruction cache cells.
  - One node read a performance counter that counted branches and incorporated it into the checksum value.
  - One node read a performance counter that counted instructions executed and incorporated it into the checksum value.
  - One node sampled the timestamp counter probabilistically one out of every 4096 times it was invoked and incorporated it into the generated 32-bit random identifier.
  - The remaining sixteen nodes read a byte of memory, added it to the 32-bit checksum value and advanced the memory pointer. A compact linear feedback shift register algorithm was used to advance the pointer in a pseudorandom pattern that was sure to cover the entire 16MB virtual space.

To invoke the test we booted the appropriate kernel on an Entity comprised of a 133MHz Pentium CPU with 32MB of RAM and an RTL-8139B 100Mb/sec NIC. The system was on the same Ethernet segment as the Authority in order to discount the effects of network latency. When the kernel invoked the challenge negotiation code, the Authority recorded that the Entity was able to receive the test, compute the checksum value and random identifier, encrypt the results and return them via the network in 7.93 seconds. Repeated invocations of the same test, either via reboot or via secure initiation by the Authority, consistently produced the same checksum result (and a different random identifier). It is worth

noting here that the encryption step takes approximately 900000 CPU cycles, or about 0.007 seconds, so it does not contribute significantly to the response delay.

In order to successfully masquerade as a Pentium 133, an attacker would be required to simulate the testcase execution flow, page tables, PTE update, ITLB and DTLB CAMs, data and instruction caches, as well as keep track of the count of different types of instructions executed.

As an initial test, we compared the native execution time against an instruction simulator. We used Bochs 2.0 [31], a program that simulates an entire x86 system. It also emulates a NE2000 network adaptor and encapsulates traffic through its host. This conveniently allowed us to use it as though it were a real system on the network. As an initial test, we configured Bochs to ignore the WRMSR and RDMSR instructions in the testcase. The resulting test was executed by Bochs in 18.53 seconds on the 2.4GHz host. Clearly, modifying Bochs to support the hardware monitoring facilities necessary for successfully running the real test to produce a correct result would increase its execution time by a considerable amount.

We then built an ideal simulator that relied on the assumption that it would have perfect *a priori* knowledge of how the test code would act. This would be equivalent to a virtualizing simulator that could *instantly* transform the testcase into an accelerated form that could run natively on the simulation host. To build such a simulator, we took each of the 22 testcase nodes and manually encoded the precise effects of their execution on the ITLB, the DTLB, caches and the two instruction counts. We also built page table traversal and update routines that worked with the TLBs. (We allowed the simulator the advantage of not having to generate a random value.) The simulator was run on a 2.4GHz Pentium 4 system that also hosted the Authority software. The execution time of the ideal simulator was 10.72 seconds. While this is not significantly more than the execution time of the native CPU, it is also based on a few unrealistically optimistic assumptions. In particular, the simulator is using *a priori* knowledge of the testcase to build a native representation with no time penalty. A real simulator would need additional time to analyze and convert the instruction stream. Furthermore, our testcase also represents an intentionally-impaired test for which it was relatively easy to construct an efficient simulator. Several improvements for the Pentium genuinity test are possible such as using self-modifying code, reconfiguring the performance counter registers on-the-fly to periodically monitor other events and using CPU registers to inspect partial information about its own instruction decoding. Incorporating this information into the testcase would require a corresponding simulator that was significantly more difficult to build.

After finding the best-case execution time of a simulator, we can use it as the maximum allowable time for the genuinity test for real Entities. We conservatively mandate that all Entities must respond within this time, regardless of their CPU speed. Since we cannot predict or measure the network delay, it is included in the total response time. This means that Entities that experience pathological network delays during test negotiation will fail. If average network delay becomes a significant component of the total turnaround time of the genuinity tests, some modifications can be made to the test generator to compensate. For instance, if the size of the virtual space mapped by the test is doubled, the execution time for the test will approximately double as well, making the network delay a smaller component of the total time.

Newer implementations of the x86 CPUs have advantages not only in terms of speed but also architectural complexity. For instance, a Pentium/MMX is significantly harder to simulate than a Pentium running even at the same speed because of much higher associativity of its TLBs as well as data and instruction caches that are double the size and associativity of its predecessor. Even though most newer CPUs have TLBs and caches larger than earlier models, this quality does not improve their ability to masquerade as earlier models or more easily support simulators that implement the earlier TLBs and caches. As we consider higher-performance x86 microprocessors, because there is a smaller margin between their performance and that of the best available simulation host, the feasibility of constructing a fast simulator is diminished. This, in conjunction with the general difficulty of constructing a correct simulator (at any speed), creates a reasonable certainty of the veracity of the genuinity test.

Finally, although we can estimate how well a virtualizing simulator might perform, it is important to remember that, presently, no such simulator exists. The certainty of whether a Pentium 133 can be unambiguously verified as genuine will depend on monitoring the state of the art in simulators.

5.4 Effectiveness of Random Number Generation

The random identifier generated during the genuinity test must be reasonably unguessable by an attacker to prevent replay attacks and interposition attacks against the protocol. To assess the effectiveness of the random

value generation, we ran a captive testcase on a single machine repeatedly and collected and analyzed the results. (In each repetition, the same checksum result was generated.) The particular testcase interrogated the timestamp counter (TSC) only 40 times. We further constrained the testcase to clear the TSC at the beginning of each repetition and limited the entropy gatherer to incorporate only the lower two bits of each of the 40 samples. Each sample was incorporated into the random value by rotating the random value by one bit and XORing the lower two least-significant bits of the TSC. This meant that each bit of the resulting 32-bit random value was the result of either one or two samples of the TSC.

We collected 99778 consecutive random values. In the entire set, there were only two 32-bit values duplicated. This roughly corresponds to the expected collisions in a linear distribution (i.e. $2/99778 = 0.000020$ and $99778/2^{32} = 0.000023$). Furthermore, a two-dimensional visual analysis of the generated values did not reveal any apparent clustering.

Because the random value is not correlated to the checksum result and does not even reveal clues about the likely outcome of the next repetition of the contrived testcase, we believe it is reasonably unguessable. In normal testcases, the random value generation is much less constrained and provides an even greater degree of unpredictability.

Some systems provide hardware-based random number generators on the motherboard chipset. For instance, the Intel i810 and AMD 768 chipsets use sampled thermal data and constrained electrical noise to generate "truly-random" numbers. Interestingly, use of these devices is not widespread [22]. One particular disadvantage of chipset-based random number generators is that an attacker could conceivably watch bus transactions to read the random number in transit from the chipset to the CPU. Our genuinity test keeps generated random values only in CPU registers and is immune to such a potential attack. Recently, a microprocessor vendor has added support for random number generation on the CPU itself [22, 24].

6 Related Work

Our work is the first to deal specifically with the problem of determining whether a remote computer is a real computer. Nevertheless, there are several related projects that propose alternative methods for creating trusted remote systems.

6.1 Execution Verification

We are not the first to leverage the time delay inherent in the computational complexity of a problem to prove characteristics of an operating environment. Much work has been done in constructing programs that can check their work to ensure their integrity [9, 47] as well as making sure that they have not taken shortcuts in their execution [10, 20]. In particular, Jakobsson and Juels characterized and articulated the concept of a *proof of work* [27] that is similar to the rationale by which our checksum works. In their taxonomy, the result of our memory checksum during directed exercising of the CPU constitutes a *bread pudding protocol*—a way of reusing a hard-to-compute result for another purpose.

Monrose, Wyckoff and Rubin illustrate a system similar to ours that verified the correct execution of code on a remote Java Virtual Machine [34] to detect the possibility that the JVM had *cheated* on a calculation. An important distinction between their work and ours is that a JVM, being a simulator, is always susceptible to potential eavesdropping by its controller and its data could be manipulated at any time. Execution verification must be performed at all times for a JVM rather than once at boot time in order to confirm determinism of computational results.

Hohl [23] presented a thorough overview of the problem of malicious hosts and potential ways of addressing it. This research recognized the apparent insolubility of the authenticity problem and concentrated on establishing an environment whereby work could be sent to (possibly malicious) remote Entities in such a manner that the integrity of computation could be proven. Secrecy was maintained by using software obfuscation techniques. This system was also sensitive to the execution time of the distributed computation in order to detect potential cheating. The cost of using malicious hosts for computation was realized by the additional network bandwidth and increased execution time.

6.2 Secure Coprocessors

Several projects in industry [17, 44, 48] and academia [50] have focused on the development and use of *secure coprocessors* that guarantee their trustworthiness in a hostile environment by employing physical tamper-proofing measures as well as incorporating mechanisms that guard against loading untrusted software. Any attempt at unauthorized access to the system would destroy its contents, so as long as it still held a usable secret, it could be assumed to be secure. Such a system could then authenticate itself to another remote system

by means of public-key cryptography based on an internal secret. Here, the system demonstrates its genuinity not by proving that it is a genuine computer but by proving its *identity*. That identity must be known in advance to other systems for the secure coprocessor's security to be meaningful to a remote party. In contrast, our work allows previously unknown systems to authenticate themselves to a central authority.

6.3 Secure Bootloaders

Several projects have been dedicated to the development of *secure bootloaders* that allow a computer system to authenticate the software that it loads [6, 11, 26, 50] by using a cryptographic secret stored in a secure coprocessor or on a smartcard. These systems require the integration of a secure BIOS or special loader to guarantee that no hostile code becomes operational on the machine in question. By comparison, our work requires no firmware modifications. It is also not sensitive to the *means* by which the loaded kernel became operational. As long as the genuinity test can verify that the kernel is running and in control of the system, a secure bootloader for the kernel offers no additional benefit.

6.4 TCPA, Palladium and LaGrande

Recently, a great deal of media attention has been focussed on the Trusted Computing Platform Alliance (TCPA) [1], Microsoft's Palladium Initiative [14], and Intel's LaGrande [35]. Nevertheless, it is not always clear how the respective systems work or what they are intended for. Many publications have been made available in an attempt to correct these misunderstandings [2, 4, 5, 18, 41, 43], and we encourage the curious reader to survey them carefully. We can point out two primary differences between these systems and our work:

- None of these systems appear to provide a means of demonstrating an anonymous system's genuinity to a central authority in the way our method does. Instead, they specifically aid in the generation and manipulation of cryptographic secrets for the purpose of identity management and access control.

- Each of the aforementioned systems require the addition of some form of hardware to a participating computer in order to manipulate and hide cryptographic secrets. We are not, at this time, advocating the addition of any security or cryptography hardware to the system, nor do we rely on any static secrets that must be integrated into the computer's firmware to support evidence of its identity.

What remains to be seen is whether these technologies will provide a useful mechanism for increasing our ability to resolve the genuinity of computer systems.

7 Summary

We have implemented a method by which a remote computer system can demonstrate the genuinity of its hardware and running software to a certifying authority without the need for a trusted human intermediary. We believe such a method will be useful for enabling the aggregation of arbitrary and, potentially, anonymous systems into distributed computational clusters. To revisit our introductory example, Alice could employ such a method to allow Mallory to add trusted hosts to her network environment with an acceptably low risk of Mallory gaining unauthorized access to resources. In addition, this could be done with no other demands on Alice's time other than the initial configuration and deployment of the certifying authority.

Our implementation shows a reasonable safety margin against potential simulation attacks even when using a very slow target microprocessor. Other more direct attacks are possible, but the complexity of their undertaking may allow our implementation to be acceptable in a number of environments. We continue to develop our implementation to broaden the number of applicable microprocessors by improving the types of meta-information that can be incorporated into the checksum algorithm. To our knowledge, all high-performance microprocessors presently in production have enough measurable side-effects to support a genuinity test. Lower-performance embedded CPUs may also be valid targets of our technique—especially if their instruction sets are not common to higher-performing CPUs since this would certainly force a potential attacker to use an instruction simulator.

Acknowledgments

We thank Mike Franzen for the numerous lengthy conversations that greatly influenced the early versions of this research. Mike Shuey also provided timely criticism in the development of the network protocol and timely loan of test machines. We thank the anonymous reviewers for their many detailed and helpful remarks.

References

[1] Trusted Computing Platform Aliance. TCPA main specification. http://www.trustedcomputing.org/.

[2] Ross Anderson. TCPA / Palladium Frequently Asked Questions. http://www.cl.cam.ac.uk/users/rja14/tcpa-faq.html.

[3] AOL. The America Online Instant Messenger Application. http://www.aol.com/, 2002.

[4] William A. Arbaugh. Improving the TCPA. *IEEE Computer*, 35:77–79, August 2002.

[5] William A. Arbaugh. The TCPA; what's wrong; what's right and what to do about it. http://www.cs.umd.edu/~waa/TCPA/TCPA-goodnbad.pdf, July 2002.

[6] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. A Secure and Reliable Bootstrap Architecture. In *IEEE Symposium on Security and Privacy*, pages 65–71, May 1997.

[7] R. Bedichek. Some efficient architecture simulation techniques. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 53–64, Berkeley, CA, 1990. USENIX Association.

[8] S. M. Bellovin and M. Merritt. An attack on the interlock protocol when used for authentication. *IEEE Transactions on Information Theory*, 40(1):273–275, January 1994.

[9] Manuel Blum and Sampath Kannan. Designing programs that check their work. In *ACM Symposium on Theory of Computing*, pages 86–97, 1989.

[10] Jin-Yi Cai, Richard J. Lipton, Robert Sedgewick, and Andrew Chi-Chih Yao. Towards uncheatable benchmarks. In *Structure in Complexity Theory Conference*, pages 2–11, 1993.

[11] Paul C. Clark and Lance J. Hoffman. Bits: A smartcard protected operating system. *Communications of the ACM*, 37(11):66–94, November 1994.

[12] Christian Collberg and Clark Thomborson. On the limits of software watermarking. Technical Report 164, University of Auckland Dept. of Computer Science, August 1998.

[13] Christian Collberg and Clark Thomborson. Watermarking, tamper-proofing, and obfuscation—tools for software protection. Technical Report 170, University of Auckland Dept. of Computer Science, February 2000.

[14] Microsoft Corporation. Microsoft "Palladium": A business overview. http://www.microsoft.com/presspass/features/2002/jul02/0724palladiumwp.asp.

[15] Donald W. Davies and Wyn L. Price. *Security for Computer Networks, second ed.* John Wiley & Sons, second edition, 1989.

[16] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-11:644–654, November 1976.

[17] Joan G. Dyer, Mark Lindemann, Ronald Perez, Reiner Sailer, Leendert van Doorn, Sean Smith, and Steve Weingart. Building the IBM 4758 secure coprocessor. *IEEE Computer*, 34(10):57–66, October 2001.

[18] Paul England and Marcus Peinado. Authenticated operation of open computing devices. In *Proceedings of the 7th Australasian Conference on Information Security and Privacy*, pages 346–361. Springer-Verlag, 2002.

[19] Free Software Foundation. GNU GRUB. http://www.gnu.org/software/grub/grub.html, 2003.

[20] Philippe Golle and Ilya Mironov. Uncheatable distributed computations. In *CT-RSA*, pages 425–440, 2001.

[21] D. Grover. The protection of computer software: Its technology and applications. In *The British Computer Society Monographs in Informatics*, chapter Program Identification. Cambridge University Press, second edition, 1992.

[22] Mark Hachman. Via adds crypto to "Nehemiah", plans mobile launch. http://www.extremetech.com/article2/0,3973,838362,00.asp, 2003.

[23] Fritz Hohl. Time limited blackbox security: Protecting mobile agents from malicious hosts. *Lecture Notes in Computer Science*, 1419:92–113, 1998.

[24] VIA Technologies Inc. The VIA Padlock Data Encryption Engine. http://www.via.com.tw/en/viac3/padlock.jsp, 2003.

[25] VMware, Inc. The VMware workstation simulator. http://www.vmware.com/, 2002.

[26] N. Itoi, W. A. Arbaugh, J. McHugh, and W. L. Fithen. Personal secure booting. In *Proceedings of the Sixth Australian Conference on Information Security and Privacy*, pages 130–144, July 2001.

[27] Markus Jakobsson and Ari Juels. Proofs of work and bread pudding protocols. In *IFIP TC6 and TC11 Joint Working Conference on Communications and Multimedia Security (CMS '99), Leuven, Belgium*. Kluwer, September 1999.

[28] Steven Kent and Randall Atkinson. Security architecture for the internet protocol. IETF RFC 2401, November 1998.

[29] John B. Lacy, Donald P. Mitchell, and William M. Schell. CryptoLib: Cryptography in Software. In *UNIX Security Symposium IV Proceedings*. USENIX Association, 1993.

[30] Kevin Lawton. The Plex86 simulator. http://plex86.org/, 2002.

[31] Kevin Lawton. Bochs: The Open Source IA-32 Emulation Project. http://bochs.sourceforge.net/, 2003.

[32] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Adreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, February 2002.

[33] Peter S. Magnusson and Bengt Werner. Some efficient techniques for simulating memory. Technical Report R94-16, Swedish Institute of Computer Science, August 1994.

[34] Fabian Monrose, Peter Wyckoff, and Aviel D. Rubin. Distributed execution with remote audit. In *ISOC Network and Distributed System Security Symposium*, pages 103–113, 1999.

[35] Paul Otellini. Intel developer forum, fall 2002, keynote speech. http://www.intel.com/pressroom/archive/speeches/otellini20020909.htm.

[36] Etherboot Project. Etherboot. http://www.etherboot.org/, 2003.

[37] Niels Provos. Encrypted virtual memory. In *Proceedings of the 9th USENIX Security Symposium*. USENIX Association, 2000.

[38] PyxisSystemsTechnologies. AIM/oscar protocol specification: Section 3: Connection management. http://aimdoc.sourceforge.net/faim/protocol/section3.html, 2002.

[39] Ronald L. Rivest and Adi Shamir. How to expose an eavesdropper. *Communications of the ACM*, 27(4):393–395, April 1984.

[40] John Scott Robin and Cynthia E Irvine. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *Proceedings of the 9th USENIX Security Symposium*. USENIX Association, 2000.

[41] David Safford. The need for TCPA. http://www.research.ibm.com/gsal/tcpa/why_tcpa.pdf, October 2002.

[42] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network Filesystem. In *Summer 1985 USENIX Conference*, 1985.

[43] Seth Schoen. Palladium details. http://www.activewin.com/articles/2002/pd.shtml.

[44] Sean W. Smith, Elaine R. Palmer, and Steve Weingart. Using a high-performance, programmable secure coprocessor. In *Financial Cryptography*, pages 73–89, 1998.

[45] A.M Turing. On computable numbers: With an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936.

[46] L. van Doorn, G. Ballintijn, and W. A. Arbaugh. Design and implementation of signed executables for linux. Technical Report HPL-2001-227, University of Maryland, College Park, June 2001.

[47] Hal Wasserman and Manuel Blum. Software reliability via run-time result-checking. *Journal of the ACM*, 44(6):826–849, 1997.

[48] Steve R. White and Liam Comerford. ABYSS: A trusted architecture for software protection. In *IEEE Symposium on Security and Privacy*, pages 38–51, 1987.

[49] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *Measurement and Modeling of Computer Systems*, pages 68–79, 1996.

[50] Bennet Yee. *Using Secure Coprocessors*. PhD thesis, Carnegie Mellon University, May 1994.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits

- Free subscription to *;login:*, the Association's magazine, published six times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and Open Source, book and software reviews, summaries of sessions at USENIX conferences, and Standards Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to *;login:* on the USENIX Web site.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, on the USENIX Web site.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as security, Linux, Internet technologies and systems, operating systems, and Windows—as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Savings on a variety of products, books, software, and periodicals: see *http://www.usenix.org/membership/specialdisc.html* for details.

USENIX & SAGE Thank Their Supporting Members

USENIX Supporting Members

❖ Ajava Systems, Inc. ❖
❖ Aptitune Corporation ❖ Atos Origin B.V. ❖
❖ Computer Measurement Group ❖
❖ Freshwater Software ❖ Interhack Corporation ❖
❖ The Measurement Factory ❖ Microsoft Research ❖
❖ Motorola Australia Software Centre ❖
❖ Sun Microsystems, Inc. ❖
❖ Taos: The Sys Admin Company ❖
❖ UUNET Technologies, Inc. ❖
❖ Veritas Software ❖ Ximian, Inc. ❖

SAGE Supporting Members

❖ Freshwater Software ❖
❖ Microsoft Research ❖
❖ Ripe NCC ❖

For more information about membership, conferences, or publications,
see *http://www.usenix.org/*
or contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA
Phone: 510-528-8649 Fax: 510-548-5738 Email: *office@usenix.org*

ISBN 1-931971-13-7